The Dissertation Committee for Pei-Chi Huang
certifies that this is the approved version of the following dissertation:

Real-Time Robotic Tasks for Cyber-Physical Avatars

Committee:

Aloysius K. Mok, Supervisor

Risto Miikkulainen

Peter Stone

Luis Sentis

Song Han

Chien-Liang Fok

Real-Time Robotic Tasks for Cyber-Physical Avatars

by

Pei-Chi Huang, B.E.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

Dedicated to my lovely family.

Acknowledgments

First and foremost, I would like to acknowledge my advisor, Professor Aloysius K. Mok for his guidance, support and encouragement throughout the research. Prof. Mok's keen insight, professional knowledge, timely advice, and enthusiasm benefit me greatly for my work. My deepest gratitude and appreciation to him is far beyond my ability to express in words.

Completing a Ph.D. is a long and hard journey, but I have been fortunate enough to have the mentors, my lab mates, my colleagues, my friends and my family during the graduate studies:

First, I would like to say thank you for Joseph Ho, thanks for providing me with a unique opportunity to obtain industrial experience that gained a wider breadth of my research and realized the differences between theoretical and practical knowledge. I truly admire your enthusiasm for research and business. I also would like to acknowledge the members of my thesis oral exam committee, Prof. Risto Miikkulainen, Prof. Peter Stone, Prof. Luis Sentis, Prof. Song Han, Dr. Chien-Liang Fok for their precious suggestions and advice.

I would like to thank my lab mates in my research group at the University of Texas at Austin, including Bing Ai, Jianyong Meng, Lixun Zhang, Quan Leng, Prof. Song Han, Wei-Ju Chen, Dr. Xiuming Zhu, Yi-Hung Wei,

Yumeng Ling, Yi-Hsuan Hsieh, and Zheng Li, for the fruitful ideas and valuable discussions. Also, I would like to acknowledge my colleagues in Hong Kong and China for sharing with the knowledge, experience, and ideas that inspired my research. It was my pleasure to work with you. I also want to thank you to the collaborators from the mechanical engineering department that have helped greatly for me to conduct experiments and obtain the performance results. I also would like to say thank you to the research director, Mark Nixon, and Dr. Deji Chen at Emerson Process Management for your leadership and guidance.

The acquaintances and friends who offer spiritual support and precious memories, helped me to get through the variety of challenges in my life, and are too numerous to name–I offer my profound thanks. Last but not least, I am grateful to my family, especially, my sibling, Kai-Yu Huang and Chin-Yin Huang, and my parents, Mao-Hsin Huang and Hsiu-Chin Chen. Without their understanding, love, support and encouragement, I cannot finish my dissertation. Words cannot express how much I appreciate what you have done for me.

Pei-Chi Huang

The University of Texas at Austin
May 2017

Real-Time Robotic Tasks for Cyber-Physical Avatars

Pei-Chi Huang, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Aloysius K. Mok

Although modern robots can perform complex tasks using sophisticated algorithms that are specialized to a particular task and environment, creating robots capable of completing tasks in unstructured environments without human guidance (e.g., through teleoperation) remains a challenge. In this research, we present a framework to meet this challenge for a "cyberphysical avatar," which is defined to be a semi-autonomous robotic system that adjusts to an unstructured environment and performs physical tasks subject to critical timing constraints while under human supervision. This thesis first realizes a cyberphysical avatar that integrates three key technologies: (1) whole body-compliant control, (2) skill acquisition from machine learning (neuroevolution methods and deep learning), and (3) vision-based control through visual servoing. Body-compliant control is essential for operator safety because avatars perform cooperative tasks in close proximity to humans; machine learning enables "programming" avatars such that they can be used by non-experts for

a large array of tasks, some unforeseen, in an unstructured environment; the visual servoing technique is indispensable for facilitating feedback control in human avatar interaction. This thesis proposes and demonstrates a systematically incremental approach to automating robotic tasks by decomposing a non-trivial task into stages, each of which may be automated by integrating the aforementioned techniques. We design and implement the controllers for two semi-autonomous robots that integrate three key techniques for grasping and pick-and-place tasks. While a general theory is beyond reach, we present a study on the tradeoffs between three design metrics for robotic task systems: (1) the amount of training effort for the robots to perform the task, (2) the time available to complete the task when the command is given, and (3) the quality of the result of the performed task. The tradeoff study in this design space uses the imprecise computation model as a framework to evaluate specific types of tasks: (1) grasping an unknown object and (2) placing the object in a target position. We demonstrate the generality of our integration methodology by applying it to two different robots, *Dreamer* and *Hoppy*. Our approach is evaluated by the performance of the robots in trading off between task completion time, training time and task completion success rate, in an environment similar to those in the recent Amazon Picking Challenge.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction to Robotic Task Design for Cyber Physical Avatars

1.1 Introduction

Although modern robots can perform complex tasks competently through hand-designed algorithms, it remains challenging to create robots capable of completing mission-critical tasks in unstructured environments without dependence on human guidance (e.g. through teleoperation) [74]. A conspicuous example of this challenge is Amazon's use of autonomous robots to fetch customers' orders in the company's massive warehouses. In line with the goal of improving warehouse automation, Amazon organized the competition, Amazon Picking Challenge[1], that challenged participants to develop their own hardware and software for the general task of picking a subset of products from inventory shelves and then placing them on a nearby table, called a *pick-and-place* task. Amazon created the competition because commercially viable automated pick-and-place tasks are still difficult. Another example of this challenge is the labor-intensive component placement and in-

[1] Amazon Picking Challenge: `http://amazonpickingchallenge.org/`

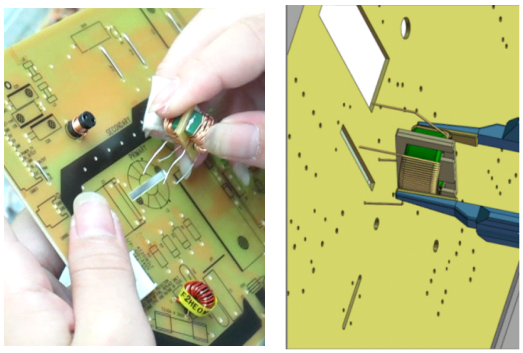


Figure 1.1: Automatic component placement and insertion assembly. The example shows the insertion of a 4-pin choke coil into circuit board holes. (a) Hand-form is labor-intensive; (b) Robot-form is cost-effective. Clearly, there are enormous expenditures for hiring workers; the longer they spend on the assembly, the higher the cost will be. The conclusion is that developing robots to replace humans in factory assembly is necessary.

sertion assembly in mass industrial production, called *pick-and-insert* task, for example, inserting a 4-pin choke coil into circuit board holes, as illustrated in Figure 1.1. Although efforts to automate pick-and-insert assembly have progressed from fully manual and product-specific programmable to full automation in industrial robotics, state-of-the-art techniques still cannot satisfy customers' demand for low-cost automation with respect to sophisticated and customized components. Each awkward or oddly shaped component must still be hand-programmed for in-line assembly through specialized robotic systems, making automation less flexible and prohibitively expensive for smaller lots.

In this research, we present a framework to meet this challenge based upon the concept of a "cyberphysical avatar," defined to be a semi-autonomous

remote robotic system that adjusts to an unstructured environment and performs physical tasks subject to critical timing constraints while under human supervision [23]. This thesis first realizes a cyberphysical avatar that integrates three key technologies: (1) whole body-compliant control, (2) skill acquisition from machine learning (neuroevolution methods and deep learning), and (3) vision-based control through visual servoing. Body-compliant control is essential for operator safety because avatars perform cooperative tasks in close proximity to humans; the prioritized whole-body compliant controller (WBC) is used for our purpose [78]. Machine learning technique enables "programming" avatars such that they can be used by non-experts for a large array of tasks, in an unstructured environment; we apply a neuroevolution approach [42], i.e. evolving an artificial neural network (ANN) with an evolutionary algorithm (EA). In particular, the widely-used NeuroEvolution of Augmenting Topologies (NEAT); [81] algorithm is one of the most powerful for tasks where the optimal behavior to be learned is unknown, but needs to be discovered by exploration. Also, we apply deep learning algorithms for object detection, recognition and its region-of-interest, which uses region proposal network built by several fully connected convolutional neural network (CNN). Vision-based control is capable of "guiding" the motion of avatars to exploit an unknown environment by using computer vision data. In the research, the visual servoing technique [32] is indispensable for facilitating feedback control in human avatar interaction.

This thesis proposes and demonstrates a systematically incremental approach to automating robotic tasks by decomposing a non-trivial task into

stages, each of which may be automated by integrating the aforementioned techniques. We design and implement the controllers for two semi-autonomous robots that integrate three key techniques for grasping and pick-and-place tasks. In these tasks, our approach has five stages. In the first stage, the human supervisor can choose to define or automatically generate a bounding box around the target object by means of the man-machine interface, In the second stage, the robot's vision system recognizes the object and determines its location, based solely on three-dimensional (3D) vision-based object features as extracted by the robot's sensors. In the third stage, the robot through off-line machine learning derives the robotic hand configuration that makes it possible to accomplish the grasping task. In the fourth stage, two controllers are applied: a visual servoing (VS) controller controls the motion of a robot to safely move the object to the destination by using the knowledge of the object's shape and of the camera's motion; a whole-body-control (WBC) controller is used to adapt the robot's body to the actual physical conditions while performing the grasping task. Finally, the robots place the object into the bin.

While a general theory is beyond reach, we also present a study on the tradeoffs between three design metrics for robotic task systems: (1) the amount of training effort for the robots to perform the task, (2) the time available to complete the task when the command is given, and (3) the quality of the result of the performed task. The tradeoff study in this design space uses the imprecise computation model [48, 49] as a framework to help explore the boundary region of tolerance and find best effort techniques. We evaluate

specific types of tasks: (1) grasping an unknown object and (2) placing the object in a target position. It is important to recognize the fact that the quality of robotic task performance is a function of at least two parameters: the amount of training the robot has had through machine learning algorithms, as well as the tightness of the real-time task deadline that the robot is to meet. For example, if we give the robot one second to grasp an unknown object, it is likely that the grasp will not be as firm and reliable as what the robot would be able to achieve given ten seconds to complete the grasp. The goal of our research is to perform a systematic investigation of the tradeoffs between the training effort, the resulting quality of the robotic task, and the time the robot takes to perform the task. Understanding this tradeoff is essential to design robots that can function effectively in real time. We demonstrate the generality of our integration methodology by applying it to two different robots, (1) *Dreamer*, a humanoid torque-controlled mobile robot in the Human Centered Robotics Laboratory at the University of Texas at Austin (UTA), and (2) *Hoppy*, an industrial robotic arm in the Cyber-Physical System Laboratory at UTA, as our experimental platform. Our approach is evaluated by the performance of the robots in trading off between task completion time, training time and task completion success rate, in an environment similar to those in the recent Amazon Picking Challenge.

1.2 Synopsis

We review previous machine learning approaches to robotic grasping and the imprecise computation techniques that are used to analyze performance. Then, we present an architectural overview of one cyber-physical system platform. We also describe the three techniques for attaining robot collaborative skills for dynamic control of humanoid avatars in unstructured environments, skills acquisition through machine learning and vision-based control of the end-effector in unknown environments. We integrate these techniques into a system for programming cyberphysical avatars. By carefully assigning and coordinating basic skills, we discuss how humanoid robotic components (e.g., hands and legs) may be integrated to accomplish more complex human-like behaviors in real-world scenarios. Finally, a case study based on the Amazon Challenge competition application is used to evaluate real-time performance tradeoffs.

The rest of this dissertation is organized as follows. Chapter 2 reviews related work and describes previous machine learning approaches to robotic grasping and imprecise computation. Chapter 3 describes system integration and its architecture. Chapter 4 presents the three techniques for achieving the research tasks. Chapter 5 addresses one robot control design flow. Chapter 6 describes training and testing experimental results and their evaluation. In Chapter 7, we discuss the results that contribute to the development of a systematic approach for designing a robotic task system that can function in environment like the Amazon Challenge competition. Chapter 8 evaluates the

real-time grasping performance by using the imprecise computation model on our robotic task system. Finally, Chapter 9 concludes this dissertation by reviewing remaining problems and lists avenues for future work.

Chapter 2

Background and Related Work

This chapter reviews previous machine learning approaches to robotic grasping, the neuroevolution method applied in the experiments, a simulation environment GraspIt!, imprecise computation techniques that are applied to analyze performance, and the Robot Operating System (ROS) framework for writing robot software in Gazebo 3D simulation.

2.1 Robotic Grasping through Machine Learning

Over the last decade, multiple robotic manipulation systems have been developed that apply motion planning approaches to generate stable robotic grasps. Such approaches generally adopt control models specifically calibrated for a particular robot in a specific environment [40] and consequently they often prove fragile when deployed in unforeseen environments. Importantly, machine learning offers the potential to overcome such limitations by applying general methods for automated learning. For example, Miller et al. [54] applied heuristic rules to produce and evaluate grasps for three-fingered hands through a fixed set of primitives (e.g. spheres, boxes, cones, and cylinders) and Pelossof et al. [59] employed Support Vector Machines (SVM) to evaluate grasp quality.

However, in both methods full two-dimensional (2D) or three-dimensional (3D) models of the target objects are given *apriori* to the algorithms, so these approaches focused only on control and planning.

Due to the limitations imposed by perception, methods of extracting visual information are also prominent approaches: One conceptually simple approach is to identity the grasp configuration. For example, Piater [60] and Coelho et al. [11] used K-means clustering to measure 2D hand orientation. Another approach is to find an optimal control policy through iterative adaptive process while performing a grasp. This approach considers the uncertainties in the environment. For example, Kamon et al. [36] controlled an arm to approach and then grasp through Q-learning that learned errors. Recently, impressive progress has been made in learning to grasp novel objects [74, 73, 75, 62, 41, 76]. The fundamental principle is to track 2D and 3D information through computer vision (e.g. objects' shapes and segmentation). Tracking this information can be then generalized to grasp novel objects.

Other approaches use reinforcement learning techniques to explore optimally when searching for control strategies [37]: Zhang and Bössler [92] defined a reward function according to simple geometrical features of objects to learn a grasp controller for a PUMA robot. Saxena et al. [74] used supervised learning to learn correct 3D grasping points for a gripper from visual features of objects. However, most of these methods assume that the current state of the system is completely known, which is often untenable in unstructured environments. While many studies indicate that robot learning from demonstration (LfD) [3]

is a promising way to improve grasping performance, complete automation of the grasping task in unforeseen circumstances remains difficult. Herzog et al., [25] created simpler heuristic-based grasp planners based on simple object parameterizations. Hsiao et al., [29, 28] partitioned hand configuration spaces into several regions, mapping into several states in partially observable Markov decision processes (POMDP) to choose optimal control policies for two-fingered hands and then used LfD to teach the robot grasping. Yet the method assumed full knowledge of the objects' 3D models and was not tested in the real world, and much more work is required to extend to complicated objects. Also, transferring controllers from simulation to reality is challenging [34, 47]; because many studies only generate simulated results [8, 67], it is likely that without modification such resulting controllers will not function robustly on a physical robot.

Related to the approach described here are previous Artificial Neural Networks (ANNs) approaches that simulate arm kinematics. For example, Rezzoug and Gorge [67] proposed two separate ANNs that respectively learn finger inverse kinematics and appropriate arm configuration, with results obtained only in simulation. Pedro et al. [55, 58] proposed that contact points can be calculated through computational geometry (e.g. Delaunay triangulation and Voronoi diagrams) before a robot performs grasping action; however their work only considered object geometry problems.

In the next section, we shall describe NEAT neuroevolution algorithm.

2.2 Neuroevolution

Neuroevolution is an approach where an evolutionary algorithm is applied to learn the structure of an ANN, its connection weights, or both [81]. Compared with other machine learning methods, neuroevolution is unique in two main ways.

First, most other learning methods are supervised, i.e., they learn behavior that approximates a given set of examples [24]. It is important that such examples are carefully chosen to ensure that the training process results in learning a function that smoothly interpolates between them. For instance, in robotic grasping, a training set consists of grasping situations paired with the corresponding optimal grasping behavior. Because optimal behavior is often not known, it is unclear how such examples can be produced to cover representative situations well. In contrast, neuroevolution is a reinforcement learning method, and as such it does not require training examples where ideal behavior is known. Instead, a measure of performance (e.g. grasp quality) is optimized by the underlying evolutionary algorithm. Second, neuroevolution does not rely on complete state information. Other methods that are designed to learn under sparse reinforcement, such as Q-learning (or value function learning in general) often assume that the current state of the system is completely known [85]. However, if objects are occluded or the situation varies dynamically, it is difficult for such methods to differentiate between possible situations because the observed values of actions cannot be associated with the correct underlying state. Neuroevolution solves the problem by evolving

recurrent connectivity; recurrence establishes memory that make it possible to distinguish between states.

One complication in applying neuroevolution to a complex domain like robotic grasping is that the ideal network topology (i.e. how many neurons compose the network and how are they interconnected) is not a known *apriori*. Because the depth image input contains many low-level features (i.e., pixels), a fully connected network with many hidden neurons may have an intractable number of parameters to tune. This motivates the NeuroEvolution of Augmenting Topologies (NEAT; [81]) method which is a popular method for evolving both network topology and connection weights. With NEAT, the ideal network topology needs not be known *apriori*, but is discovered automatically as part of evolution. The NEAT method was originally developed to evolve ANNs to solve difficult control and sequential decision tasks [81, 82]. Evolved ANNs control agents that select actions based on their sensory inputs. NEAT begins evolution with a population of small, simple networks and *complexifies* the network topology into diverse species over generations, leading to increasingly sophisticated behavior. To keep track of which gene is which while new genes are added, a historical marking is uniquely assigned to each new structural component. During crossover, genes with the same historical markings are aligned, producing meaningful offspring efficiently. Speciation in NEAT protects new structural innovations by reducing competition among differing structures and network complexities, thereby giving newer, more complex structures room to adjust. Networks are assigned to species based on the

extent to which they share historical markings. Complexification, which resembles how genes are added over the course of natural evolution, is thus supported by both historical markings and speciation, allowing NEAT to establish high-level features early in evolution and then later elaborate on them. A more complete description of NEAT algorithm can be found in [81].

In the following section, we shall introduce grasping simulation.

2.3 GraspIt! Overview

To apply neuroevolution to learn where and how to grasp an object requires both training scenarios and a measure for evaluating performance. GraspIt! [53] is an interactive simulation, planning, analysis, and visualization tool for robotic grasping. Given this simulator, we need an effective training method to learn grasping behaviors automatically. We select the NEAT neuroevolution learning method because it has shown previous promise both in grasping [5] and in extracting features from low-level input [38].

Next, we present the concept of imprecise computation.

2.4 Imprecise Computation

Imprecise computation is a scheduling technique that reduces the amount of time used on a job by means of sacrificing levels of quality of service (QoS) [48, 49]. In real-time applications (e. g., safety-critical applications), if the best desired quality of results cannot be obtained, imprecise

computation decreases the QoS to make it possible to meet timing constraints of real-time tasks while still keeping the quality within an acceptable range. In the method, each time-critical task (or a set of tasks) is divided into two parts: executing mandatory subtasks produces imprecise results that satisfy the minimum QoS requirement, whereas executing optional subtasks can enhance the imprecise quality results as time permits. In the research, imprecise computation technique helps us explore the boundary region of tolerance and find best effort techniques. Suppose that a grasping task can be divided into the mandatory parts and the optional parts. The mandatory part includes balancing *Dreamer*'s body, obstacle avoidance, and timing constraints; the optional part includes trajectory based on motion planner. Our output is to evaluate the quality of each grasp. With an increased number of subsequent deadline constraints, the objective of the experiments is to derive the relationship between time unit and grasping trajectory accuracy. To quantify the effect of trading off grasping quality for guaranteed tasks deadline, a metric is needed to measure both quality and time.

In the next section, the environment and framework of the robot are introduced.

2.5 Gazebo with Robot Operating System Overview

To integrate the techniques in a more realistic environment, we adopt a robust and high graphical quality robot simulation - Gazebo[1] which is an open source robotic simulation integrated with the Robot Operating System (ROS) framework [61]. Gazebo uses the open source OGRE rendering engine which produces good graphics fidelity. ROS is also an open source that aims to implement a flexible and extensible framework for writing robot software in robotics research which provides a communication layer on top of host operating systems to support large-scale robot software development and integration. Gazebo with ROS provides us robot manipulation and range sensing and physical properties of the robots.

The architecture of the system is described in the next section.

[1]Gazebo, `http://www.gazebosim.org/`

Chapter 3

The Semi-Autonomous Robotic System

Having summarized the motivation for designing cyberphysical avatars, emphasizing the important contributions that they could make, we turn to present the actual architecture of a cyberphysical avatar[1], also called a semi-autonomous robotics system, used interchangeably in this thesis.

This semi-autonomous robotic system comprises a mobile dexterous humanoid robot *Dreamer* with its whole body control system, and devised machine learning algorithms (Neuroevolution) including awareness of the environment complexity and sensing unpredictable world, and a real-time physical distribution network, and a series of cost-effective, real-time and vision system. The specific task explored in this work is controlling the *Dreamer* robot to approach and pick up a designated target object under remote human supervision in a real-time environment. The physical realization of the cyberphysical avatar has been implemented in the Human Centered Robotics Laboratory (HCRL) at UTA [86], and the portable remote control user interface is located in another building nearby. To provide bandwidth guarantees

[1]This chapter is previously published in [31]. I contributed the system design, integration, and implementation to our work.

for reliable real-time communication between the avatar and its human user, OpenFlow switches are being deployed within the UTA campus network.

Figure 3.1 illustrates an overview of the semi-autonomous robotic system. *Dreamer* consists of a torso, two arms, two hands, an anthropomorphic head [78]. The *Dreamer* is equipped with torque and sensors to provide force compliant capabilities. A desktop PC running Ubuntu Linux with the RTAI Real-time Kernel executes the models and control infrastructure to govern *Dreamer*'s behavior via EtherCAT serial ports. Two types of cameras are installed in the system. A Kinect camera connects to a laptop and is installed in front of the robot to capture images and depth information, and an IP camera is installed at the ceiling to capture *Dreamer*'s surrounding environment. The Kinect laptop connects to the avatar and sends images to the remote supervisor.

To apply neuroevolution to learn where and how to grasp an object requires both training scenarios and a measure for evaluating performance. GraspIt! [53] is an interactive simulation, planning, analysis, and visualization tool for robotic grasping.

A grasping experiment is achieved as follows. First, the human supervisor directs the *Dreamer* robot with a command to grasp the desired object. The cyberphysical avatar communication software relays the human input and depth information to a neural network that has been evolved with NEAT. Recall that NEAT's role is to train a neural network in a simulator to produce the appropriate outputs for *Dreamer* to act on. To apply NEAT to learn where

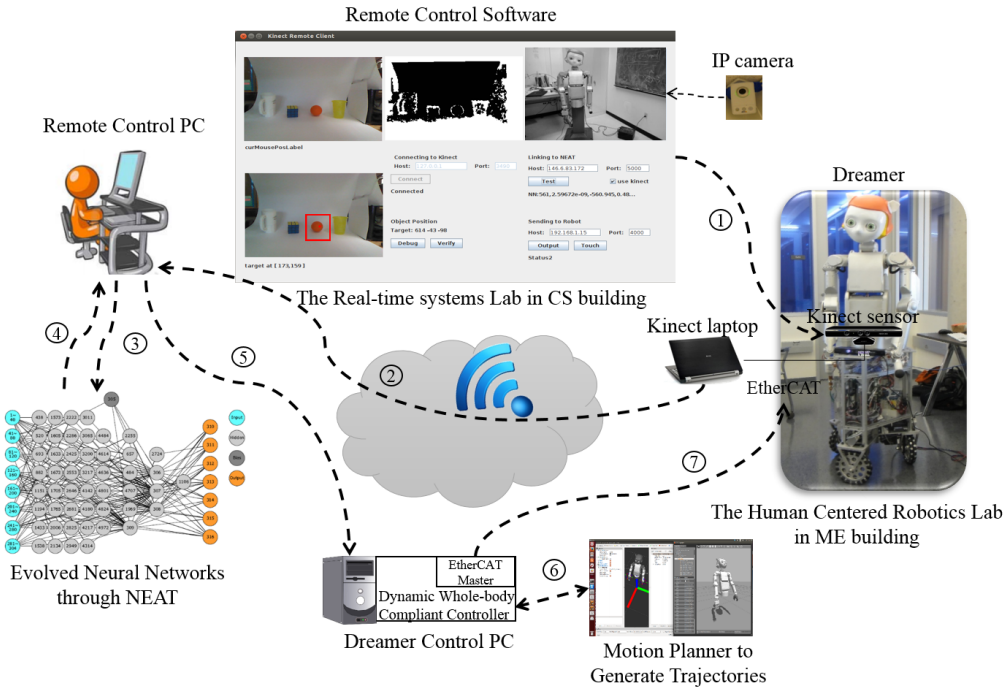


Figure 3.1: The semi-autonomous robotic system. (1) A human supervisor connects to the Kinect laptop, (2) captures a depth image, and (3) parses the depth array to serve as input to an evolved Artificial Neural Network (ANN). (4) The neural network's output is interpreted as directions to control *Mekahand*'s position and orientation, and is sent to the supervisor. (5) The supervisor sends commands to manipulate *Dreamer* robot. (6) Motion planner generates a trajectory from the initial state to the final state. (7) The controller of the wheeled humanoid avatar controls its body and arm to destination in unstructured environments. The conclusion is that the system integrates real-time vision, neuroevolution as a training method, and control manipulator while skillfully reaching a object through the man-machine interface.

and how to grasp an object requires both training scenarios and a measure for evaluating performance. GraspIt! [53] provides the interactive simulation, planning, analysis, and visualization. The neural network (trained off-line) outputs the appropriate positions and orientations to *Dreamer* robot which then moves towards the destination and grasps the targeted object with its *Mekahand*.

Next, we present the three techniques for the collaborative skills.

Chapter 4

Techniques for the Collaborative Skills

Having summarized the motivation and the architecture for designing our system, we now discuss three key techniques that have been developed (some by us) for achieving our research goal. Section 4.1 introduces the dynamic model control of humanoid avatars in unstructured environments, developed by Prof. Luis Sentis, and and the *Mekahand* model. Section 4.2 presents machine learning techniques that constitute some powerful tools for reducing expert knowledge required for autonomous control. Section 4.3 describes a vision-based control of the end-effector in unstructured environments. We integrate these techniques into a system for programming cyberphysical avatar[1].

4.1 Dynamic Control of Humanoid Avatars in Unstructured Environments

Ideally, a robot should be capable of performing the physical tasks while performing accurate physical whole-body compliant interactions both with the

[1]This chapter is previously published in [30]. I contributed the algorithm design, system integration, and implementation to our work.

environment and its human operators. To attain the capability, skill modeling and control in unstructured environments must be carefully designed. In this section, we describe the methodology for modeling the dynamic behavior of upper body based on Prof. Luis Sentis' research. We combine whole-body compliant controller (WBC) with other tools, including visual servoing and controllers based on neuroevolution learning.

Dreamer's upper body consists of 3-DOF torso, 7-DOF arms and a 12-DOF *Mekahand*, as shown in Figure 4.1. The 3-DOF torso has one unactuated joint which is coupled with the waist joint. The hand also has five actuated joints and seven coupled unactuated joints, shown in Figure 4.1. To simplify the controller, we divided the controller into one for controlling the body and the arm, and the other for controlling the hand.

To control the body and the arm together, skill modeling and dynamic control of the robot is necessary. The prioritized whole-body compliant controller (WBC) is used for our purpose [78]. In WBC, first an objective is set and then a task is defined by a Jacobian [77] to derive the relations between the robot's 10-dimensional joint spaces and the M-dimensional operational space. The controller is derived from the following constrained system dynamics equations.

$$A\ddot{q} + b(q, \dot{q}) + g(q) + J_c^T \lambda = U^T T, \tag{4.1}$$

where A is the mass matrix of the system, q is the joint coordinate vector, b is the torque caused by Coriolis and Centrifugal effects, g is the torque

caused by gravity, J_c is the constrained Jacobian, λ is the Lagrangian multiplier that describes the constrained joints, U is the actuation matrix, and T is the torque input to the system. The reason why the constrained Jacobian and the Lagrangian multiplies are shown in the system is to model the underactuated torso and the transmission constraint. The body joints 1 and 2 are coupled together. Therefore, we can specify the constraint as follows:

$$\dot{q}_1 - \dot{q}_2 = 0, \tag{4.2}$$

$$J_c \dot{q} = 0, \tag{4.3}$$

$$J_c = \begin{bmatrix} 0 & 1 & -1 & 0 & \cdots & 0 \end{bmatrix} \in \mathbf{R}^{1\times 10}. \tag{4.4}$$

We can take the constrained mass matrix Λ_c, the dynamically consistent generalized inverse of J_c, and the constrained null space N_c to derive the constrained dynamic equation as follows:

$$\Lambda_c \triangleq \left(J_c A^{-1} J_c^T\right)^+, \tag{4.5}$$

$$\overline{J_c} \triangleq A^{-1} J_c^T \Lambda_c, \tag{4.6}$$

$$N_c \triangleq I - \overline{J_c} J_c, \tag{4.7}$$

$$\ddot{q} = A^{-1} N_c^T U^T T. \tag{4.8}$$

Then, we can define task space specifications to derive the desired forces in the constrained dynamic systems. In the case of the position task that makes the end-effector (hand) approach the object, the task Jacobian is defined as

$$\dot{x} = J_{\text{position}} \dot{q}, \tag{4.9}$$

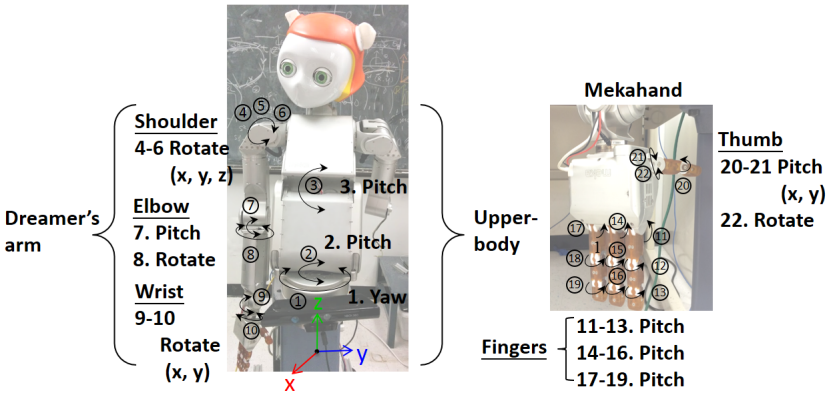


Figure 4.1: *Dreamer*'s upper body and the *Mekahand*. *Dreamer* contains 3-DOF torso (1-3), a 7-DOF arm (4-10) and a 12-DOF *Mekahand* (11-22). Each unactuated/actuated joint is coupled with another joint. The conclusion is that since many DOFs increase in difficulty of *Dreamer*'s balance control while grasping, it is necessary to design a skill modeling and dynamic control of *Dreamer*.

where x is the end-effector coordinate. The task Jacobian can describe the relation between the joint velocities and the coordinate system that a supervisor expects to control. The task Jacobian does not include the constrained dynamics, so we need to project this Jacobian to the constrained space and then generate the constrained task Jacobian,

$$J^*_{\text{position}} \triangleq J_{\text{position}}\overline{UN_c}. \tag{4.10}$$

The grasping skill, including posture, position and orientation, is defined as a juxtaposition of multiple operational tasks to help translate between high-level goals, such as those provided by the planning algorithms, and the operational tasks. In the robot's environment, a skill is composed of the three tasks in Figure 4.2: hand position, hand orientation, and the posture of

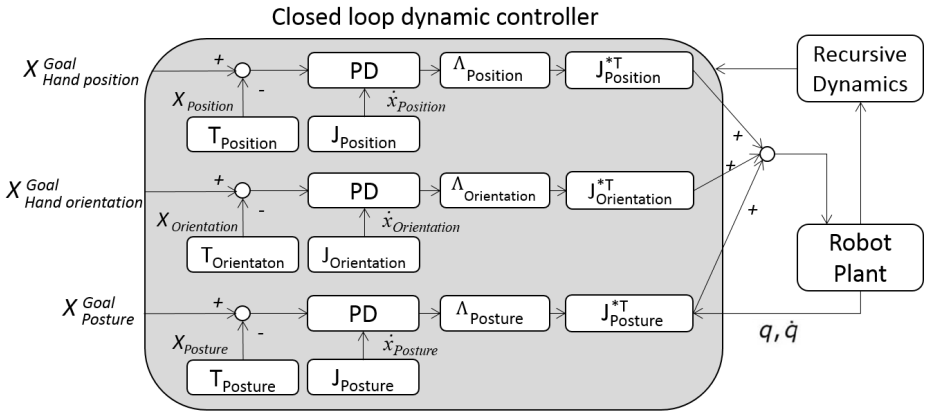


Figure 4.2: The designed whole-body compliant controller (WBC). The tasks of hand position, hand orientation and the posture of *Dreamer* upper-body are combined to perform a grasping skill. The feedback/feedforward control policies contributes to the closed-loop dynamic controller. The conclusion is that the designed control structure can effectively utilize dynamic and contact models of the physical robot in unstructured environment.

Dreamer's whole body. The control structure can be expressed as

$$\tau_{\text{control}} = J^{*T}_{\text{position}} F_{\text{position}} + J^{*T}_{\text{ori}} F_{\text{ori}} + J^{*T}_{\text{posture}} F_{\text{posture}}, \tag{4.11}$$

where F_{position}, F_{ori} and F_{posture} are the force or impedance commands to control the hand, and J^{*T}_{position}, J^{*T}_{ori} and J^{*T}_{posture} are the whole-body task Jacobians [77]. The grasping process consists of three behaviors, among which the *Mekahand* position and orientation have higher priority than the whole-body posture task because the latter is of secondary concern in the context of grasping.

The feedback control policies for the entire controller are shown in Figure 4.2, which depicts the closed-loop dynamic controller. In the figure, the Goal Hand Position/Orientation is a compliant hand position/orientation that enables the robot to reach a designated position/orientation. The Goal

Posture control exploits the remaining DOFs to stabilize self-motions. The proposed feedback/feedforward control laws are

$$\begin{aligned} F_{\text{position}} &= \Lambda^*_{\text{position}}(-k_{p,\ \text{position}} e^{\text{goal}}_{\text{position}} - k_{v,\ \text{position}} \dot{x}_{\text{position}}) \\ &\quad + p_{\text{position}}, \end{aligned} \tag{4.12}$$

$$F_{\text{orientation}} = \Lambda^*_{\text{ori}}(-k_{p,\ \text{ori}} e^{\text{goal}}_{\text{ori}} - k_{v,\ \text{ori}} \dot{x}_{\text{ori}}) + p_{\text{ori}}, \tag{4.13}$$

$$\begin{aligned} F_{\text{posture}} &= \Lambda^*_{\text{posture}}(-k_{p,\ \text{posture}} e^{\text{goal}}_{\text{posture}} - k_{v,\ \text{posture}} \dot{x}_{\text{posture}}) \\ &\quad + p_{\text{posture}}, \end{aligned} \tag{4.14}$$

where $\Lambda^*_{\text{position}}$, Λ^*_{ori} and $\Lambda^*_{\text{posture}}$ are the inertial matrices projected in the manifold of the constraints, $e^{\text{goal}}_{\text{position}}$, $e^{\text{goal}}_{\text{ori}}$ and $e^{\text{goal}}_{\text{posture}}$ are feedback error functions, k_p, k_v are gain matrices, and p_{position}, p_{ori} and p_{posture} are gravitational terms. This structure is a derivation of the previous work on compliant whole-body control [78].

Since our designed control structure can effectively use dynamic and contact models of the physical robot in its environments, it is able to optimize the process of approaching and grasping objects simultaneously, and to achieve precise tracking of forces and trajectories within the contact conditions. Thus, the grasping skill is acquired through neural network described next.

4.2 Skills Acquisition through Machine Learning

Although robots can be often controlled through carefully hand-designed algorithms, this thesis proposes one way in which reinforcement learning methods can provide a significant advantage: optimization of robot behav-

iors. The difficulty in designing effective control algorithms by hand suggests that machine learning may be a desirable approach, yet to apply common supervised learning algorithms requires a corpus of labeled examples. In contrast to supervised learning, a measure of quality is sufficient to apply reinforcement learning algorithms. In many cases, it is much easier to derive a measure of how desirable a particular behavior is than it is to either hand-construct that behavior or provide a comprehensive corpus of optimal example behaviors.

For this reason, our approach applies reinforcement learning to facilitate learning high-level behaviors that can then be invoked by a human operator. In particular, neuroevolution algorithms have proven effective in domains with low-level continuous features, which are characteristics of the problem here, i.e. learning to grip objects given depth sensor information. This section introduces our proposed approach, which is based on applying the popular neuroevolution learning method NEAT and HyperNEAT to the GraspIt! simulation environment. Here, we use the grasping skill to illustrate our idea. 4.2.1 introduces the grasping learning approach; 4.2.2 then describes the learning process, specifying the input and output layers, as well as fitness function, and also how to speedup the process.

4.2.1 Grasp Learning Approach

Our approach takes inspiration from Kohl et al. [38] who showed that neuroevolution can develop effective automobile warning systems from only low-level sensor input (i.e. pixels) taken from a digital camera. A similar

vision-based feature extraction approach is applied here, where through neuroevolution the *Mekahand* robotic arm learns appropriate hand positions and orientations for grasping. Such learning is enabled by interacting with objects in the GraspIt! simulation environment, which is described next followed by the approach to measure grasping quality and determine a visual bounding box for grasping.

4.2.1.1 GraspIt! Simulation Implementation

To apply neuroevolution to learn where and how to grasp an object requires both training scenarios and a metric for evaluating performance. GraspIt! [51, 68] facilitates simulating the *Mekahand* robot in representative grasping tasks and aids in measuring the quality of resulting grips.

GraspIt! only provides a rough *Mekahand* model, so we extended the simulator to better model it. In GraspIt!, the *Mekahand* is defined by one DOF for each knuckle in each finger, with an additional DOF for the thumb's rotator. The mechanics of this model are modified here to augment two aspects of the simulation. First, controlling the wrist is not modeled by default, but is an important DOF. Therefore, a wrist component was added to the *Mekahand* model supplied by GraspIt!. Second, most of the DOFs in the real *Mekahand* are not actuated, although they are modeled as actuated in the GraspIt! simulation. Each finger of the real *Mekahand* consists of three joints that are all connected by a single rubber tendon. Thus when the finger curls, all three knuckles curl in unison. Therefore, the torques in GraspIt! were adjusted such

that the set of torques given to a single finger are equivalent to the torques initiated by stretching the rubber tendon in the real robot.

GraspIt! uses a quaternion to represent the rotation of a 3D object. Since our learning output applies axis-angle representation in a 3D Euclidean space. Our implementation automatically translates the quaternion into the axis-angle representation in a 3D Euclidean space for the output.

4.2.1.2 Grasp Quality Measure

An evolutionary search optimizes a fitness function that measures the quality of candidate solutions. Because robust grasping behaviors are desired in this experiment, an important consideration is how to measure the quality of grasps appropriately.

Much previous grasping quality research focuses only on contact types and positions, ignoring hand geometry and kinematics. Other measures assume simple grippers. In contrast, Miller and Allen [52] proposed a more sophisticated approach, which is used here. Given a 3D object and posture of the hand, their measure can accurately identify the types of contact points between the links of the hand and the object and compute the grasp's quality. Their approach was adopted in GraspIt! to measure grasp quality of the *Mekahand*, and is described below.

Let $\dot{c}_i$, $1 \leq i \leq \dot{n}$, denote a set of contact points used to grasp an object, with each contact given a Coulomb friction with coefficient μ. In Figure 4.3(a), the applicable contact force $\dot{F}$ must certainly lie within a friction cone that

has a $\dot{F}_\perp$ with half-angle $\tan^{-1}\mu$. In Figure 4.3(b), the nonlinear cone $\dot{F}$ is approximated by a pyramid i with $\dot{m}$ sides. A unit grasp force $\dot{f}_i$ is represented as a convex combination of $\dot{m}$ force vectors [52]:

$$\dot{F}_i \approx \sum_{j=1}^{\dot{m}} \dot{\alpha_{ij}} \dot{f}_{ij},\ \dot{\alpha_{ij}} \geq 0,\ \sum_{j=1}^{\dot{m}} \dot{\alpha_{ij}} = 1, \tag{4.15}$$

where $\dot{f}_{ij}$ denotes the jth force vector around pyramid i, and $\dot{\alpha_{ij}}$ are convex weights. Assume that a reference point $\dot{r}$ is the object's center of gravity, that grasp forces $\dot{F}_i$ acting at a contact point on the object create the torque $\dot{\tau_i}$, and that these forces and torque vectors can be concatenated to form a wrench, which is given by

$$\dot{w}_i = \begin{bmatrix} \dot{F}_i \\ (\dot{c}_i - \dot{r}) \times \dot{F}_i \end{bmatrix}. \tag{4.16}$$

The grasp matrix $\dot{W}$ is formed by assembling a set of wrenches $\dot{w}_i$ for $\dot{n}$ contacts.

$$\dot{W} = \begin{bmatrix} \dot{w}_1 \dot{w}_2 \dots \dot{w}_n \end{bmatrix}. \tag{4.17}$$

One grasp applied to an object, is defined as wrenches that are capable of keeping the object in static equilibrium when

$$\dot{W}_c = -\dot{w_{ex}}, \tag{4.18}$$

where c is the vector of contact wrench magnitudes and $\dot{w_{ex}}$ is the external disturbance wrench.

The properties of the convex hull of the applied wrenches were utilized to measure grasp quality. Ferrari et al. [56, 15] proposed one method of finding

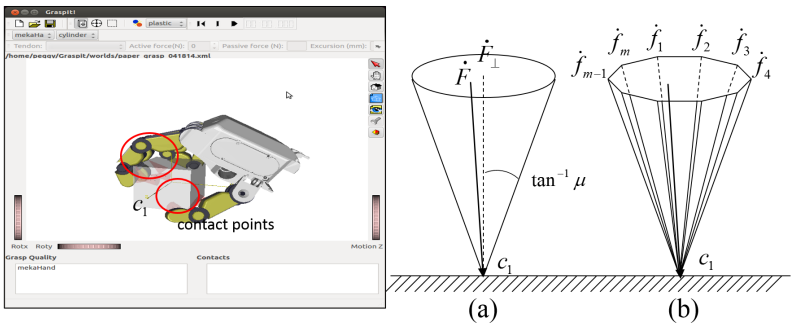


Figure 4.3: Measuring grasp quality. (a) A single contact point in 3D illustrating the friction cone with half-angle $\tan^{-1}\mu$. (b) A unit grasp force $\dot{F}_i$ is represented by the convex combination of $\dot{m}$ vectors. This quality metric can be utilized to score each grasp for machine learning.

the unit grasp wrench space, so the set of wrenches $\hat{W}$ can be applied to the object:

$$\hat{W} = \text{ConvexHull}(\dot{W}). \tag{4.19}$$

If the convex hull contains the wrench space origin and no external disturbance force, then the grasp is stable. In the convex hull, the distance from the origin of wrench space to the nearest facet can thus be used as a quality metric for the grasp [52].

The next Subsection shows how this approach can be applied to a human-supplied bounding box and focusing the robot's visual processing on the target object, thereby lessening the dimensionality of the robot's computer vision processing.

4.2.1.3 Neuroevolution

Neuroevolution is an approach whereby an evolutionary algorithm is applied to learn the structure of an ANN, its connection weights, or both [81]. One complication in applying neuroevolution to a complex domain like robotic grasping is that the ideal network topology (i.e. how many neurons compose the network and how are they interconnected) is not known *a priori*. Because the depth image input contains many low-level features (i.e., pixels), a fully connected network with many hidden neurons may have an intractable number of parameters to tune. A popular method to solve this problem is the NeuroEvolution of Augmenting Topologies (NEAT; [81]) approach for evolving both network topology and connection weights. With NEAT, the ideal network topology needs not be known *a priori*, but is discovered automatically as part of evolution. A more complete description of NEAT algorithm can be found in [81]. However, many neuroevlution methods such as NEAT are *direct-encoded*, which means each component in the phenotype is encoded by a single gene, making the discovery of repeating motifs expensive and improbable [43]. One improved *indirect-encoded* method for neural networks is the Hybercube-based NeuroEvolution of Augmenting Topologies (HyperNEAT) [33], an extension of NEAT approach. HyperNEAT has proven effective in robots control domains, including many-joint robot arm control [89], and quadruped locomotion [10]; a complete introduction can be found in Stanley et al. [80]. The following sections, we will only use NEAT to illustrate our idea.

4.2.1.4 Visual Bounding Box

In the experiment, ANNs through exploration learn how to grasp objects by integrating information from a high-dimensional depth image provided by a Kinect sensor. To better focus on the most important features of the depth image, a bounding box strategy was implemented. For each object extracted from the original scene, image data was considered only from within a supervisor-specified bounding box. The bounding box thus serves to minimize the number of irrelevant pixels considered and then simplifies the learning problem.

The training process with the bounding box method proceeds as follows. GraspIt! loads a scene, and then two mouse clicks from the user specify a rectangular bounding box that encompasses the object. In the simulated implementation, because all relative 2D coordinates of each object can be determined, an encompassing bounding box is automatically generated and centered on the desired object. For simplicity, all the computed bounding boxes have the same size. The boundary range can be mapped to four coordinates. For example, in Figure 4.4, a cube is chosen, so the bounding box is $(C_x, C_y), (C'_x, C_y), (C_x, C'_y), (C'_x, C'_y)$. The depth array of the bounding box is then divided into $M \times N$ pixels that are given to the ANN being evaluated as input data.

To simplify the implementation, the position of the camera sensor is always set such that the origin O_{3d} $(0, 0, 0)$ in the GraspIt! scene is in the center of the 2D plane, as shown in Figure 4.4. Because the input is reduced

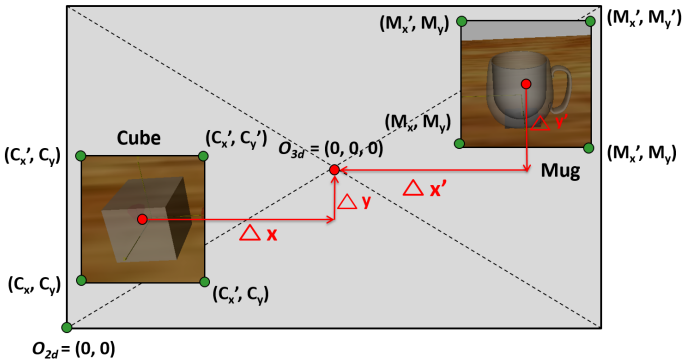


Figure 4.4: Bounding boxes of a cube and mug, and the output shift offsets Δx and Δy ($\Delta x'$ and $\Delta y'$). Because all relative 2D coordinates of each object are known, an encompassing bounding box is generated centered on the desired object. This figure shows that the boundary range can be mapped to four coordinates. To simplify implementation, the position of the camera sensor is always set such that the origin O_{3d} $(0, 0, 0)$ in the GraspIt! scene is always in the center of 2D plane. The conclusion is that a bounding box strategy can focus on the most important features of the depth image.

to a small part of the overall depth image, after the ANN produces the output, the position of each object must be offset relative to the bounding box. For example, in Figure 4.4, for the cube, Δx and Δy should be added to the position of the output, for mapping to the normalized origin position.

4.2.2 Learning Process

Combining neuroevolution with the grasping task requires specifying the input and output layers of the neural network, as well as a fitness function to evaluate grasps. A schematic description of the general framework combining GraspIt! and NEAT is depicted in Figure 4.5. Note that there are no supervised examples that the algorithm attempts to emulate. Instead, the algorithm learns from reinforcement feedback based on only the measured

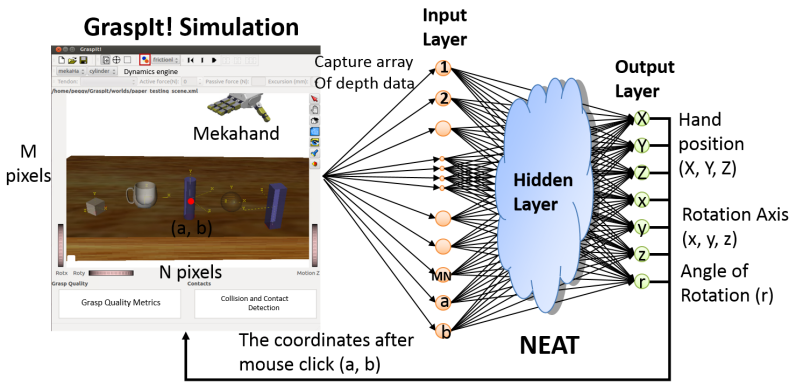


Figure 4.5: Representation of the designed grasp controller network. The left side of the figure shows GraspIt! simulation environment; the right side of the figure shows a neural network receiving input consisting of depth data and the goal coordinate (a, b) on the GraspIt! visual input scene. The network has seven output nodes: hand position (X, Y, Z), rotation axis (x, y, z) and rotation angle (r). Note that NEAT can add internal hidden nodes as evolution progresses. The figure shows how to implement grasping experiments with NEAT in GraspIt!.

quality of attempted grasps. In this way, evolution can discover solutions that work well even when the optimal behaviors are unknown.

Each ANN evaluated by NEAT receives input data denoting the current state of the robot in its environment. It is thus necessary to encode such state information, which includes the position of the target object as well as information about the object's shape. To eliminate dependency on high-level human-provided features of the grasped object, the object's state is described only by general low-level features provided by a depth map. In particular, each pixel in the depth information array is assigned a unique input node, as shown in Figure 4.5. In this way, the network can potentially learn to associate the state of an arbitrary object in an arbitrary environment with an appropriate grasping strategy.

Each ANN predicts where the object is and in what direction to grasp the object by outputting 3D hand positions and orientations. Note that each dimensional coordinate of the *Mekahand*'s position and orientation maps to one output neuron. Because the orientation is expressed in an axis-angle format (e.g. a 3D axis vector and one angle), the total dimensionality is seven, i.e. the ANN has seven output neurons. Evolution is initialized with ANNs with input nodes that are fully connected to at least a single hidden neuron, and with the hidden node fully connected to the output neurons. Recall that during evolution, ANNs can accumulate additional connections and nodes through structural mutations that augment network topology.

4.2.2.1 Grasping Fitness Function

A key element of the experimental design is to construct a fitness function to guide the search process for an appropriate ANN grasp controller. The design of a fitness function is a critical factor for guiding successful evolution.

In particular, in this experiment, the fitness of a network n with respect to an object O has four components:

- f_1: Grasp quality metric Q, described in 4.2.1.2.
- f_2: The reciprocal of Euclidean distance $d(\overrightarrow{P_i}, \overrightarrow{O_i})$ between the hand position computed by the neural network ($\overrightarrow{P_i}$) and a desired object ($\overrightarrow{O_i}$). Note that $\overrightarrow{P_i}$ and $\overrightarrow{O_i}$ are vectors.
- f_3: The reciprocal of Euclidean distance $d(\overrightarrow{P_i}, \overrightarrow{S_i})$ between the hand

position computed by the neural network ($\overrightarrow{P_i}$) and the actual hand co-ordinate after interacting with the environment ($\overrightarrow{S_i}$). Note that $\overrightarrow{P_i}$ and $\overrightarrow{S_i}$ are vectors.

- f_4: An angle θ between the *Mekahand* and grasping object. Let $\overrightarrow{V_1}$ be one vector from the center of the palm to the fingertip of the thumb; let $\overrightarrow{V_2}$ be the vector from the hand position to the center-of-gravity of the desired object; let $\overrightarrow{V_3}$ be the vector indicating the direction of the hand's axis of rotation. Let θ_1 (θ_2,respectively) be an angle between $\overrightarrow{V_1}$ and $\overrightarrow{V_2}$ ($\overrightarrow{V_2}$ and $\overrightarrow{V_3}$, respectively). To ensure that the center of palm always turns toward the object, the sum of θ_1 and θ_2 must be roughly around 90°. Figure 4.6(a) is one good case where the hand axis-angle is almost perpendicular to the object. Figure 4.6(b) is one bad case where the palm of hand is not orientated toward the object. Here, $\overrightarrow{V_1}$ and $\overrightarrow{V_2}$ ($\overrightarrow{V_2}$ and $\overrightarrow{V_3}$, respectively) are normalized so that $\|\overrightarrow{V_1}\| = \|\overrightarrow{V_2}\| = 1$ ($\|\overrightarrow{V_2}\| = \|\overrightarrow{V_3}\| = 1$, respectively). The angle θ is the sum of θ_1 and θ_2 as follows:

$$\theta = \theta_1 + \theta_2 = \theta(\overrightarrow{V_1}, \overrightarrow{V_2}) + \theta(\overrightarrow{V_2}, \overrightarrow{V_3}) \tag{4.20}$$

$$= \frac{\arccos(\overrightarrow{V_1} \cdot \overrightarrow{V_2})}{\| \overrightarrow{V_1} \| \| \overrightarrow{V_2} \|} + \frac{\arccos(\overrightarrow{V_2} \cdot \overrightarrow{V_3})}{\| \overrightarrow{V_2} \| \| \overrightarrow{V_3} \|}. \tag{4.21}$$

Thus, the fitness function f of a network n is defined as follows:

$$f = f_1 + f_2 + f_3 + f_4 \tag{4.22}$$

$$= \gamma Q + \frac{\beta}{d(\overrightarrow{P_i}, \overrightarrow{O_i}) + \alpha} + \frac{\lambda}{d(\overrightarrow{P_i}, \overrightarrow{S_i}) + \epsilon} + f(\theta). \tag{4.23}$$

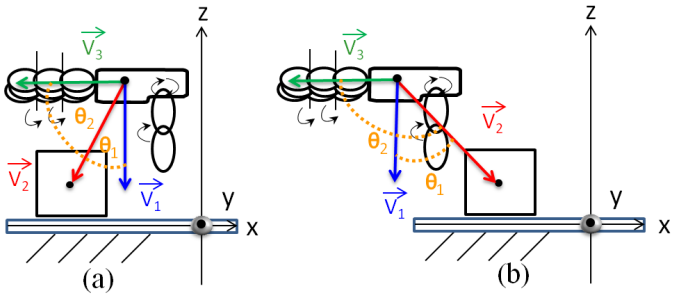


Figure 4.6: An angle θ between the *Mekahand* and grasping object. $\overrightarrow{V_1}$ is a vector from the center of palm to the fingertip of the thumb; $\overrightarrow{V_2}$ is a vector from the center of palm to the center-of-gravity of the cube; $\overrightarrow{V_3}$ is a vector from the *Mekahand*'s rotation axis. (a) A good case where the palm's center is facing the target object; the sum of θ_1 and θ_2 is almost 90°. (b) A bad case where the palm's center is not facing the target object; the sum of θ_1 and θ_2 is larger than 90°. The conclusion is that because the center of palm facing towards a object can increase the grasping opportunity, the component was added to reward the fitness function.

where α, β, γ, λ and ϵ are constants chosen to balance the various parameters.

Note that

$$f(\theta) = f(\theta_1 + \theta_2) = \begin{cases} \omega & \text{, if } 85° \leq \theta \leq 95° \\ 0 & \text{, otherwise.} \end{cases}$$

During the initial phases of evolution, when the neural networks are mostly untrained, all networks may direct the *Mekahand* to grasp at positions where it cannot even touch the object. As a result, in early generation f_1 is often effectively zero. Thus in this stage, f_2, which rewards approaching the target object, is important for differentiating the fitness of ANNs. After further evolution, when the hand can grasp the object, f_1 begins to dominate and the neural networks are ranked mostly by grasp quality. In addition, the third term f_3 is large if the *Mekahand* is not blocked by obstacles (e.g. objects other than the target object). Finally, the fourth term (f_4) rewards facing the

palm of the robotic hand towards the target object. Parameters α, β, γ, λ, ϵ and ω adjust the relative effects of those four terms. In this way, the described fitness function rewards ANNs first to learn to approach the object, and then to grasp the object in an increasingly appropriate way. Algorithm 1 shows the fitness function in detail.

Alg 1 Computation of the Fitness Function

Input: Q is the grasp quality after the execution of a single grasp, θ is the summation of θ_1 and θ_2, $\overrightarrow{P_i}$ is the predicted position of hand for grasping by the network, $\overrightarrow{O_i}$ is the coordinate of the selected object after the mouse click, $\overrightarrow{S_i}$ is the actual hand coordinate after interacting with the environment.
Output: A fitness evaluation of a single grasp.
Let A_j be a set of 3D coordinates of objects in the environment, where $1 \leq j \leq n$;
for $j = 1$ to n **do**
    $Dist_o = \min(Dist_o, \sqrt{\sum_{i \in x,y,z} (\overrightarrow{A_{j,i}} - \overrightarrow{S_i})^2})$;
end for
$Dist_t = \sqrt{\sum_{i \in x,y,z} (\overrightarrow{O_i} - \overrightarrow{S_i})^2})$;
if $(Q = 0) \parallel (Dist_o < Dist_t)$ **then**
    {∗No grasp quality or *Mekahand* is closer to other objects.∗}
    $f_1 = 0$;
    $f_2 = \frac{\beta}{d(\overrightarrow{P_i}, \overrightarrow{O_i}) + \alpha}$;
else
    $f_1 = \gamma$Q, where $\gamma \geq 10000$;
    $f_2 = k$, where $k \leq 1000$;
end if
$f_3 = \frac{\lambda}{d(\overrightarrow{P_i}, \overrightarrow{S_i}) + \epsilon}$;
if $(85 \leq \theta)$ && $(\theta \leq 95)$ **then**
    if $Dist_t < 50$ **then**
        $f_4 = \omega$;
    else
        $f_4 = w$, $w < \omega$;
    end if
else
    $f_4 = 0$;
end if
return $sum = \sum_{i=1}^{4} fit_i$;

4.2.2.2 Reducing Training Time through Parallelization

The computational cost incurred by the *sequential* implementation of the fitness function computation is as follows. For one experiment, each generation consists of $\hat{o}$ ANNs, and each ANN is evaluated over $\hat{s}$ object combinations. Each object combination contains $\hat{b}$ objects, and each object is selected as $\hat{k}$ candidates to be an input. If one experiment runs for $\hat{g}$ generations, the total number of independent training simulations in GraspIt! T is $\hat{o} \times \hat{s} \times \hat{b} \times \hat{k} \times \hat{g}$. In our experiments, $\hat{o} = 200, \hat{s} = 5, \hat{b} = 4, \hat{k} = 10, \hat{g} = 150$. Thus, $T = 6,000,000$. Therefore, a parallel strategy that dispatches different trials to all available computer cores is implemented to encourage computational efficiency. In particular, work is dispatched over a network to multiple GraspIt! processes that run on different computers. In this way, each CPU core in different computers can be fully employed, and the resulting multi-threaded implementation speeds up the evolution process.

Figure 4.7(a) illustrates the sequential method for each generation. To reduce execution time, the following computational steps were parallelized, as shown in Figure 4.7(b). First, three commands are defined: *cmd_get_info* is to get the depth array, *cmd_get_quality* is to get the quality for each grasp, and *cmd_shuffle* is to change the position and orientation of each object. Here, assume that four instances of GraspIt! are run and waiting for commands. Two kinds of threads are created: *Organism tasks* that use ANNs from NEAT's main process to generate grasping tasks and collect the resulting fitness score; and *GraspIt! tasks* that communicate with a GraspIt! process to send the

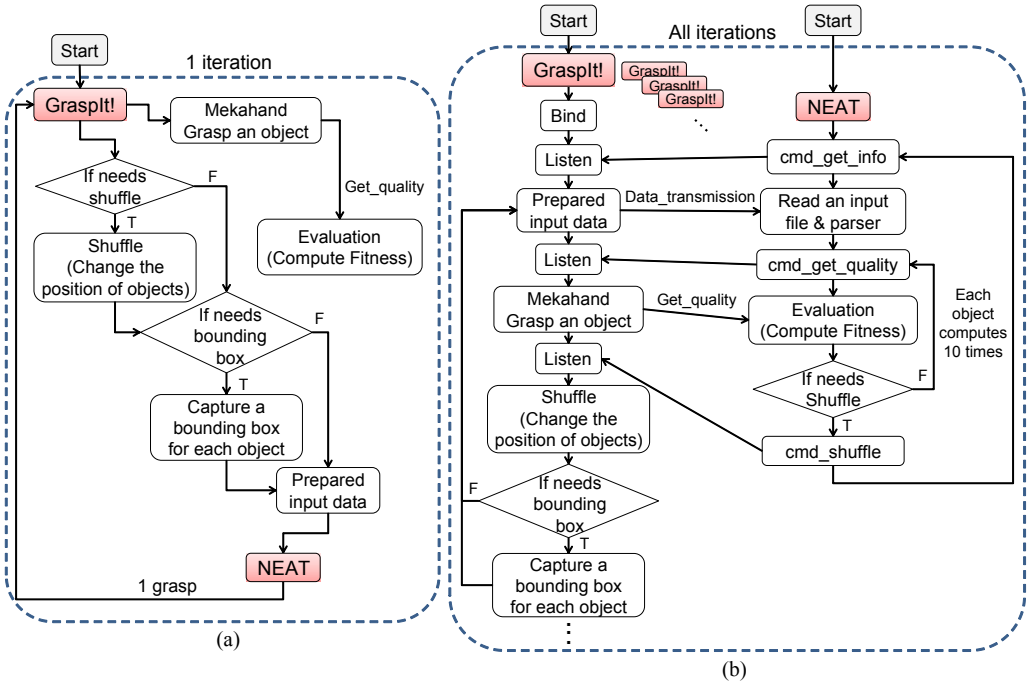


Figure 4.7: The same computers were used to compare the sequential and parallel comparison methods.(a) The original sequential method. (b) The faster parallel method. The results show that with the original sequential implementation, the program only utilizes a single core, but after parallelizing the algorithm, the program can fully utilize four cores, and the experiment's run time is shortened by a factor of three.

output from an ANN for simulation in GraspIt!, and receive the resulting grasp quality. The speedup achieved by such parallization depends on how many GraspIt! processes are run. Our results show that the run time is accelerated by at least a factor of three.

4.3 Vision-Based Control of the End-Effector in Unstructured Environments

In the proposed research, our plan is to move the robot arm to the vicinity of the target object without changing the position of the robotic base and then provide direction to achieve the end-effector alignment with the target object such that the robot can successfully perform a grasp. However, a control without any sensory feedback may easily lead to failure in unstructured environments. Therefore, vision-based controllers have been proposed to manipulate objects since the early 1980s. The robot obtained the information from vision sensors for the control software to generate a motion plan for the end-effector to reach the desired destination. The target of all vision-based control schemes is to minimize error functions.

Due to the nature of the open-loop motion planner, the trajectories generated by it may not tolerate the errors that occur close to the input sensor and mechanical uncertainties. Thus, without frequent calibrations [22] of the vision system and the robotics arms, we may not be able to make the end-effector move accurately to the desired destination. Furthermore, inherent hardware errors (e.g., wear and tear of the mechanical parts) and inefficient synchronization among sub-systems will accumulate errors and eventually will cause the end-effector to miss the destination. As opposed to the open-loop method, *visual servo control* is a close-loop method that uses feedback from image information extracted from cameras to accurately control the motion of the robot.

4.3.1 The Design of Visual Servoing

In terms of the design of visual servoing (VS) for object manipulation, three issues are considered: (1) the chosen control law, (2) camera-robot configuration, and (3) computer vision algorithms [39].

1. **The chosen control law:** Image-based visual servoing (IBVS) is adopted because IBVS is easily implemented by measuring an error signal in the image space and then directly reducing to robot motion commands. Position-based visual servoing (PBVS) requires high-precision calibration parameters and good quality of cameras, as well as the desired object model [13].

2. **The chosen camera-robot configuration:** The configuration depends on what kind of details we can provide, and what level of complexity the task is, and it also determines different types of computer vision algorithms for the feedback of the control. We choose only one monocular mounted and fixed camera configuration outside the environment such that this camera can directly acquire the end-effector's information and the object's information at the same time.

3. **The chosen algorithm of images information:** For vision data, the aim is to detect the relevant information features from raw image data to reduce representation. First, a modified Microsoft Kinect sensor or two camera sensors are used to fetch image sources to provide both RGB and depth information. Then, visual features are extracted for

each image by using a histogram of oriented gradients (HOG) or scale invariant feature transform (SIFT). The objects to be manipulated are detected and recognized by performing support vector machine (SVM) and HOG features (or SIFT features), and these combinations results in an impressive performance [14].

4.3.2 Control Law Principles

Recall that the modeling of classical VS scheme is to minimize an error function $e(t)$ [9]:

$$e(t) = s(m(t), a) - s^*, \tag{4.24}$$

where the vector $m(t)$ is a set of interest points/features of images derived by computer vision algorithms; a is additional object information, such as a 3D model or camera intrinsic parameters; $s(m(t), a)$ is a vector of f visual features; and s^* is the features values of desired object. Here, given the fixed target object and the fixed camera, the extracted features from the objects are always the same. Also, IBVS only requires a set of image features, so a in s can be ignored. The fundamental IBVS structure is shown in Figure 4.8. The motion is aligned by measuring on the errors e between the desired position of objects s^* and the current position of robots s perceived by vision. Thus, the eq.(4.24) can be simplified as $e(t) = s(m(t)) - s^*$, where s^* is constant.

Because the visual features have been selected, the simplest control scheme is to design a velocity controller. To consider the end-effector moving in the workspace, given the end-effector $\grave{e}$ and camera c frames, the end-effector

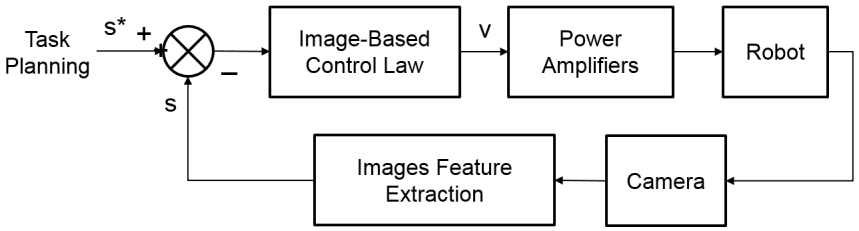


Figure 4.8: Image-based visual servo (IBVS) structure. For each planned task, IBVS control law is designed to control the robot arm in the vicinity of the object using visual feedback. Visual feedback is captured and measured to help in matching the final destination. The conclusion is that IBVS can be applied to control an end-effector of *Dreamer* that enables tracing the end-effector and bringing it to the desired position in dynamic environment.

$\mathbf{v}$ is a point that is rigidly attached to the end-effector with the camera frame. The rotate matrix represents the orientation of camera frame c with respect to end-effector frame $\grave{e}$, which is denoted by ${}^{\grave{e}}\mathbf{v_c}$. The relationship between the time variation of s and ${}^{\grave{e}}\mathbf{v_c}$ can be derived by

$$\dot{s} = L_s {}^{\grave{e}}\mathbf{v_c}, \tag{4.25}$$

where L_s is the interaction matrix related to s, also called *feature Jacobian*, used interchangeably here.

Based on eq.(4.24) and (4.25), the time variation of the error e is related to ${}^{\grave{e}}\mathbf{v_c}$ by

$$\dot{e} = L_s {}^{\grave{e}}\mathbf{v_c}. \tag{4.26}$$

Let $L_s = L_e$ and let ${}^{\grave{e}}\mathbf{v_c}$ be an input to the controller. We expect that an error for a designed control law will approach 0, as t approaches infinity (i.e., $\dot{e} = \lambda e$ with $\lambda < 0$). In order to obtain the relationship between ${}^{\grave{e}}\mathbf{v_c}$ and e, since L_e is singular, let L_e^+ be replaced by the MoorePenrose pseudoinverse of L_e under

some constraints. According to eq.(4.26) and using analyzed computation [9], the control law between $\dot{\mathbf{e}}_{\mathbf{v_c}}$ and e can be derived by:

$$\dot{\mathbf{e}}_{\mathbf{v_c}} = \widehat{L_e^+} e, \tag{4.27}$$

where $\widehat{L_e^+}$ is the approximation of the pseudoinverse of the interaction matrix.

4.3.3 Interaction Matrix

IBVS control utilizes image features provided by computer vision system as feeding information in a close-loop system, so here we describe the image formation process. Assume that the camera pose $C_{pose}$is known. A set of 2D virtual points can be projected and computed from their corresponding 3D points. Let ${}^{o}x$ be the coordinates of a 3D point in the object frame, and let ${}^{c}x = (X, Y, Z)$ be its corresponding 3D point in the camera frame. We get:

$$\begin{bmatrix} {}^{c}x \\ 1 \end{bmatrix} = C_{pose} \begin{bmatrix} {}^{o}x \\ 1 \end{bmatrix}. \tag{4.28}$$

Assume that ${}^{c}x$ projects onto an image plane as a 2D feature coordinate $p = (x, y)$, which can be expressed with:

$$\begin{cases} x = X/Z \\ y = Y/Z \end{cases} \tag{4.29}$$

We differentiate the eq.(4.29) with respect to time t, then

$$\begin{cases} \dot{x} = \dot{X}/Z - X\dot{Z}/Z^2 = (\dot{X} - x\dot{Z})/Z \\ \dot{y} = \dot{Y}/Z - Y\dot{Z}/Z^2 = (\dot{Y} - y\dot{Z})/Z \end{cases} \tag{4.30}$$

Suppose that the end-effector is moving with angular velocity ω_c and translational velocity v_c, both with respect to the camera frame. Based on [32],

the well-known equation ${}^c\dot{x} = -\omega_c \times^c x - v_c$ can be applied, and then we can derive the relationship between the velocity of the 3D point ${}^c x$ and $\dot{\mathbf{e}}_{\mathbf{v_c}}$ can be expressed as follows:

$$\begin{cases} \dot{X} = -\omega_y Z + \omega_z Y - v_x \\ \dot{Y} = -\omega_z X + \omega_x Z - v_y \\ \dot{Z} = -\omega_x Y + \omega_y X - v_z \end{cases} \tag{4.31}$$

$\dot{X}$, $\dot{Y}$, $\dot{Z}$ in eq.(4.30) can be substituted by eq.(4.31). Then, the velocity of this 2D point $\dot{p}$ can be obtained:

$$\dot{p} = \begin{cases} \dot{x} = -v_x/Z + xv_z/Z + xy\omega_x - (1+x^2)\omega_y + y\omega_z \\ \dot{y} = -v_y/Z + yv_z/Z + (1+y^2)\omega_x - xy\omega_y - x\omega_z \end{cases} \tag{4.32}$$

We can simplify $\dot{p} = Lp\mathbf{v_c}$, where Lp is the interaction matrix related to p. Assuming that m has k features ($k \geq 6$), according to eq.(4.25), we yield the velocities below:

$$\begin{bmatrix} \dot{x_1} \\ \dot{y_1} \\ \vdots \\ \dot{x_k} \\ \dot{y_k} \end{bmatrix} = \begin{bmatrix} \frac{-1}{Z_1} & 0 & \frac{x_1}{Z_1} & x_1 y_1 & -(1+{x_1}^2) & y_1 \\ 0 & \frac{-1}{Z_1} & \frac{y_1}{Z_1} & 1+{y_1}^2 & -x_1 y_1 & -x_1 \\ & & & \vdots & & \\ \frac{-1}{Z_k} & 0 & \frac{x_k}{Z_k} & x_k y_k & -(1+{x_k}^2) & y_k \\ 0 & \frac{-1}{Z_k} & \frac{y_k}{Z_k} & 1+{y_k}^2 & -x_k y_k & -x_k \end{bmatrix} \begin{bmatrix} v_X \\ v_Y \\ v_Z \\ \omega_X \\ \omega_Y \\ \omega_Z \end{bmatrix}. \tag{4.33}$$

For image processing methods, we can use scale-invariant feature transform (SIFT) and speeded up robust features (SURF) to detect and capture keypoints on images. Here, fiducial markers are attached to the end-effector, and we also use a middleware, ControlIt![2] [16], for controlling many DOF robots. By actively measuring the error, visual servoing can achieve the following tasks: (1) adapt to changes in the objects position and orientation

[2]ControlIt!, `http://robotcontrolit.com/installation`

and (2) account for modeling errors within ControlIt! This will enable the end-effector to be positioned more accurately in the required position around the object and thus increase the successful rate of picking up the object in dynamic environment.

Chapter 5

Synergy of Skills

We now describe the actual architecture of a cyberphysical avatar. We use the two terms cyberphysical avatar and semi-autonomous robot interchangeably in 5.1, but we remind the reader that the concept of a cyberphysical avatar is meant to emphasize the role of the human supervisor in intelligent robotics and the need for a pathway from teleoperation to full autonomy. In this context, an incremental design method for different tasks will be discussed for industrial product assembly. More complex assembly operations will be achieved by coordinating multiple robotic components. In section 5.2, we discuss how by carefully assigning and coordinating basic skills, humanoid robotic components (e.g., hands and legs) may be integrated to accomplish more complex human-like behaviors in real-world scenarios.

5.1 Skills Augmentation

Our experimental platform for the cyberphysical avatar is a mobile dexterous humanoid robot called *Dreamer*. *Dreamer* is controlled by a WBC controller in coordination with controllers devised by machine learning algorithms and a visual servoing system. The first specific task explored in this work is

controlling the *Dreamer* robot to approach and pick up a designated target object under remote human supervision while subject to real-time constraints. A physical realization of the cyberphysical avatar has been implemented in the Human Centered Robotics Laboratory[1] (HCRL) at UTA, and the portable remote control user interface is located in another building nearby.

Many tasks in real-world product assembly scenarios require a set of steps to be performed in sequence. The incremental approach we are proposing here will ensure that each acquired skill can be reused to achieve common factory assembly operations: the grasp task, the pick-and-place task, and the pick-and-insert task. The approach includes three features: first, four modes determine which controller we should use. Second, the three controllers from Chapter 4 perform different functions. Third, the independent modules are reusable for the three tasks mentioned above. Figure 5.1 illustrates a workflow of the general task in this semi-autonomous robotic system and is organized into several modules that direct the flow of visual information into the completion of a task, including four swapped modes under different states. The goal of switching modes is to deal with different conditions, and these modes can be applied for fulfilling various tasks.

1. **Supervisor mode:** Decide which task will be completed. This mode allows for the help from a human supervisor and benefits from knowledge acquisition. For example, the supervisor can provide a description of the

[1]Human Centered Robotics Laboratory, `http://www.me.utexas.edu/~hcrl/`

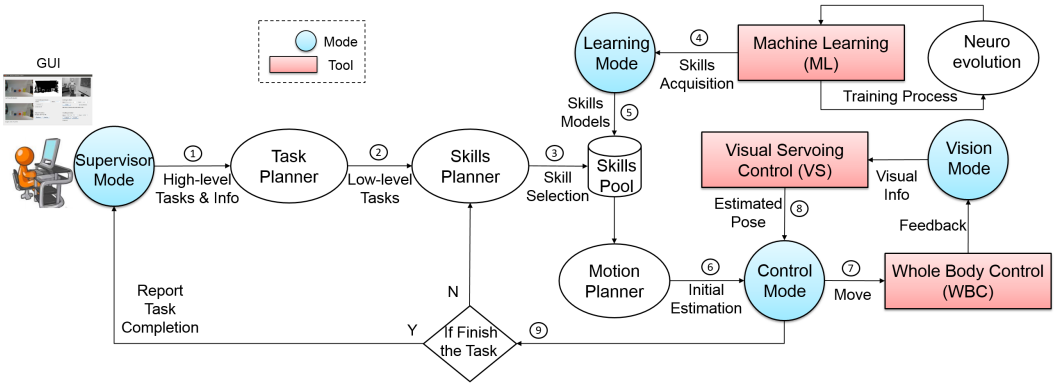


Figure 5.1: A workflow of the simple general task with the sequential method in the semi-autonomous robotic system. (1) A human supervisor provides a high-level description for a task based on the captured visual information. (2) The task planner gives low-level description and further breaks down a task into several skills. (3). The skill planner selects skill from a skill pool accordingly to be executed (4) Machine learning (e.g., neuroevolution) helps train skills models and sends the acquired skill to the skills pool. (5) The skill output through model is interpreted as directions to control the end-effector's position and orientation. (6) The motion planner generates a trajectory from the initial state to the final state. (7) The controller of the wheeled humanoid avatar controls its body and arm to approach the destination. (8) The visual servoing controller repeatedly guides the robot to approach the object. (9) The controller checks the current status and judges whether the task has been fulfilled or not. In summary, the system comprises two parts (i.e., knowledge acquisition through machine learning and task execution with integrated real-time vision and control manipulator, which skillfully reaches an object through the man-machine interface.) Also, our approach combines supervisor, vision, learning, and control modes while employing different situations. Through carefully exercising these modes and situations, a robot can then complete a task successfully.

target object or designate the component to be grasped by pointing and bracketing the object with a specified bounding box. This type of information may eventually be automated in the upper levels of a hierarchically organized intelligent system.

2. **Vision mode:** Automatically obtain visual information. The mode derives the locations of objects by applying computer vision techniques to capture/segment images and then extract features.

3. **Learning mode:** Provide a classifier and an optimized policy. The mode produces the skills pools for the use of robots through machine learning techniques (e.g., Neuroevolution) to discover the characteristics for each specific component. The input can be depth/color features extracted from the output of vision mode. The output can be continuous and represent the end-effector configurations. A reward function should be designed, including the current manipulated components state, its output results, the component's locations, and other environmental dynamics. The training set contains the image information taken from different poses of components to derive an optimized policy.

4. **Control mode:** Manipulate the components of the robot and complete the required instructions from the task planner. The mode uses a motion planner and visual servoing (VS) technique to reach an object and move while balancing the robot's body with WBC technique.

An example of achieving a grasping task in Figure 5.1 is as follows. First, the human supervisor directs the *Dreamer* robot with a command to grasp the desired object. The communication software for the system relays the human input and image/depth information to a neural network that has been evolved with neuroevolution (e.g., NEAT). Recall that NEAT's role is to train a neural network in a simulator to produce the appropriate outputs for *Dreamer* to act on. To apply NEAT to learn where and how to grasp an object requires both training scenarios and a measure for evaluating performance. GraspIt! [53] provides interactive simulation, planning, analysis, and visualization. The neural network (trained off-line) delivers the appropriate positions and orientations to the *Dreamer* robot, which then moves toward the destination and grasps the targeted object with its *Mekahand*. In the physical environment, the visual servoing technique, applied to control *Dreamer* for visually guided object grasping, first moves the robot to a location close enough to grasp the object and also makes sure the end-effector and object are in the same view frame. Then, we repeatedly servo the end-effector to a grasp configuration obtained from NEAT. Finally, the object can be successfully grasped.

The same process can be applied to the pick-and-place task. In Figure 5.1, the task planner divides this task into grasping and placing skills. After performing the grasping task at step (9), the placing task can be repeatedly executed at step (3). The skill planner also can be repeatedly executed for the pick-and-insert task until the success condition is met. For example,

we assume that the task requires the following lower-level physical skills: (1) object detection, (2) object identification, (3) object grasping, (4) hole finding, and (5) object insertion. We use Supervisor mode to decide which component will be picked up first, and then we transit to Learning mode to exploit an optimized policy and a classifier system, which deals with (1)-(3). Note that the training model can be derived from machine learning techniques. Once the optimized results are produced, the destination of whole-body arrival and an end-effectors grasping configuration are derived, and then we plan the robot's trajectory path and an end-effectors path on the basis of the nearby environment, robot's configurations, and performance models. Then, we swap to Vision mode to detect several components from the piles and estimate their approximate placements through camera sensors to perform (4). At the end, we change to Control mode to work (5). If our target is close enough to the end-effector, WBC technique will be executed to attain the goal; otherwise, visual servoing technique will be called upon to help in automating the approach toward reaching the component.

5.2 Skills Composition

Many complex tasks require cooperation among different parts of humanoid robots. For example, if we need to hammer a nail into a wall, one hand has to place and hold the nail on the wall while the other has to hammer the nail concurrently to complete the task. In order to apply different skills on each hand concurrently, all computations are scheduled and coordinated by

the task planner in a distributed fashion. Of particular interest in the area of cooperative manipulation, coordination is the skill assignment problem that allocates a given number of cooperative components to execute a series of skills as efficiently as possible. The problem can be reduced to a *task-allocation* problem. While a skills planner that assigns only for an individual component is relatively simple, the difficulty occurs when at least two skills are composed to finish a task. One possible solution that makes use of resource management is to determine an efficient assignment of skills to robots.

Finding a general plan to achieve complex tasks is an NP-hard problem. Therefore, a task is assumed to be bounded by a finite number of steps performed on the limited number of components (e.g., *Dreamer*'s two hands), and a task can be acquired from either a task planner that decomposes several skills or a supervisor that specifically describes an input. An enhanced flowchart for two components (assume left hand and right hand) to complete a collaborative task in the semi-autonomous robotic system, depicted in Figure 5.2, is added with one more module scheduler. Because it is harder to allocate multiple resources at once, the task planner in Figure 5.1 takes priority over these skills and then feeds into the skills planner and scheduler. The skills planner constructs a linked-list with every skill labeled and pointers to record the number of skills. The scheduler decides when and how to carry out the skills such that completion time can be minimized subject to the constraint that the task must be finished by a limited number of cooperative components simultaneously. Also, the scheduler is a centralized controller

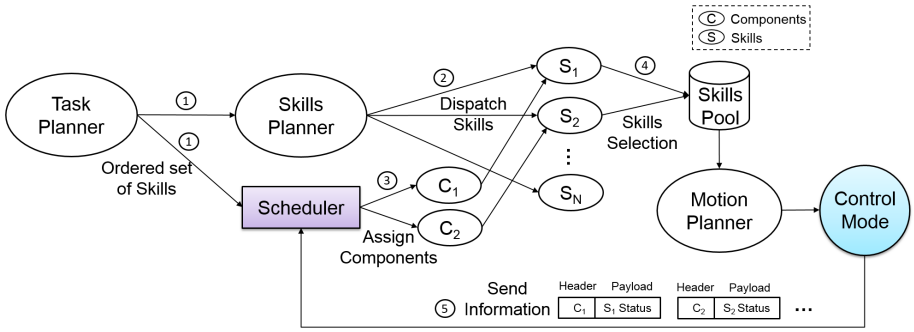


Figure 5.2: A workflow of composition tasks with the coordinated method in the semi-autonomous robotic system. (1) The task planner produces a skills order of assembly and sends it to the skills planner and scheduler. (2) The skills planner allocates the set of skills. (3) A scheduler arranges available components to the set of currently valid skills. (4) Each skill selects a corresponding model from the skills pool which is derived from machine learning techniques. (5) A controller in control mode delivers a notification to inform the scheduler of the current status of components. The sequence of steps can help robots successfully complete the composition task.

that actively communicates with the components with a TCP/IP packet like header/payload structure to deploy the requested skills to the components and receive the status response from them.

Recall the previous example of using a hammer to pound the nail into the wall. Here are the respective procedures: (1) identify the right nail, (2) grasp the nail, (3) identify the hammer, (4) hold the hammer, (5) estimate the place where the nail will go into the wall, (6) move the nail to the target location, (7) guide the hammer around the spot, (8) drill a starter hole into the wall by using the nail, (9) hold the nail tightly, (10) hammer the nail into the starter hole until the nail penetrates, (11) release the nail, and (12) place the hammer. In Vision mode, steps (1), (3), and (5) can be completed with visual

aids. The right hand performs (4), (7), (10), and (12); the left hand operates (2), (6), (8), (9), and (11). Also, (9) depends on (10). We can formalize the task allocation problem as follows: Let $S = \{s_1, s_2, \ldots, s_p\}$ be the finite set of skills, and $E = \{e_1, e_2, \ldots, e_p\}$ be the set of execution time of their corresponding skills, and $C = \{c_1, c_2, \ldots, c_q\}$ be the finite set of components. Assume that two identical controllers control the right hand (C_1) and the left hand (C_2), separately. Each skill s_i has different c_i, where $1 \leq i \leq p$. The problem can be reduced to a multiprocessor scheduling problem. We try to answer what the minimum possible total execution time ($\sum E$) required to schedule all skills is. The scheduler is a skills manager that improves the efficiency with the completion time of task by executing skills in parallel. For example, steps (1) and (2) and steps (3) and (4) can manipulate simultaneously. In that way, the system also can be compatible with heterogeneous components of robots that increase its productivity and flexibility.

Chapter 6

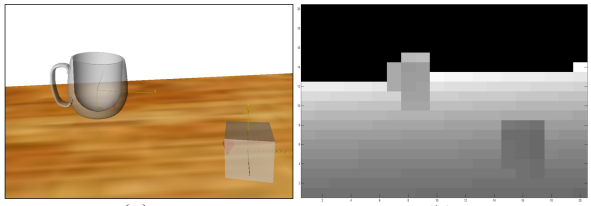
Evaluation

This chapter presents the implementation of the proposed skills and also conducted a series of experiments to evaluate the system performance[1] in Section 6.1. Then, the integration of the system from simulation to reality is introduced in Section 6.2.

6.1 Training and Testing Grasping Experiments

In this section, the training and testing experiments are described. The first set of training experiments combines the four fitness components in different ways, as described in 6.1.3. The best combination is applied in the second and third sets of training experiments, which evaluate the benefit of applying a bounding box to focus the ANN's attention in 6.1.4. Fully trained ANNs are tested in simulation (6.1.5).

[1]This chapter is previously published in [30]. I contributed the performance implementation and evaluation to our work.



(a) (b)

Figure 6.1: Sample input data for training neural networks. (a) The RGB pixel data of the scene from the camera within GraspIt!. (b) The 20×15 depth data array supplied to the neural network as input. The depth data is normalized to a floating point number between [0, 1]. The purpose is that the original raw pixel data is high-dimensional, so a down-scaled data of the same data can be easily performed in practice.

6.1.1 Experimental Design

Because the raw depth data from the Kinect sensor is of high dimensionality, for practical purposes the array is first down-scaled. Before the input data is supplied to an ANN, the 640×480 pixel array was sampled to form a reduced 20×15 array. A larger scale was also tried, such as 40×30, 80×60, but the evolution process was so time-consuming that we decided to shrink back 20×15 array. This smaller array was converted to gray-scale intensity values, and then normalized between zero and one; an example is shown in Figure 6.1. The input data also includes a coordinate that represents the mouse click input from the user that specifies the target object. In the grasping experiments, the coordinate is chosen by randomly picking a different point on the target object in each trial. To increase accuracy in evaluating each network, they are each evaluated five times over different trials. That is, the robot attempts to grasp each target object five times, and the fitness value is the average over

all the attempts. To preserve generality, the position and orientation of the objects for each evaluation are randomized.

The experiments are divided into two parts: training and testing. A collection of objects are divided into N separate classes, and for each class, ANNs are trained by NEAT to grasp objects from that class. For testing, the best neural network generated from training is further tested in simulations over objects placed in different locations. A final test applies a real scenario from *Dreamer* to the evolved neural networks. The flowchart for training and testing is shown in Figure 6.2. All experimental parameters are described in 6.1.2.

6.1.2 Experimental Parameters

In the experiments, the population size was set to 120. Different values of the three parameters α, β and γ of the fitness function (Eq. 4.22) were tried and tuned to guide evolution. The number of generations was 100. The coefficients for measuring compatibility for NEAT were $c_1 = 1.0$, $c_2 = 2.0$, $c_3 = 2.0$. The survival threshold was set to 0.2. The drop-off age was set to 20. Recurrent connections were disabled because the task is not dependent on history. The probability of adding nodes and adding new connections to evolved ANNs were set to 0.2 and 0.3, respectively. Detailed description of these parameters are given in [81].

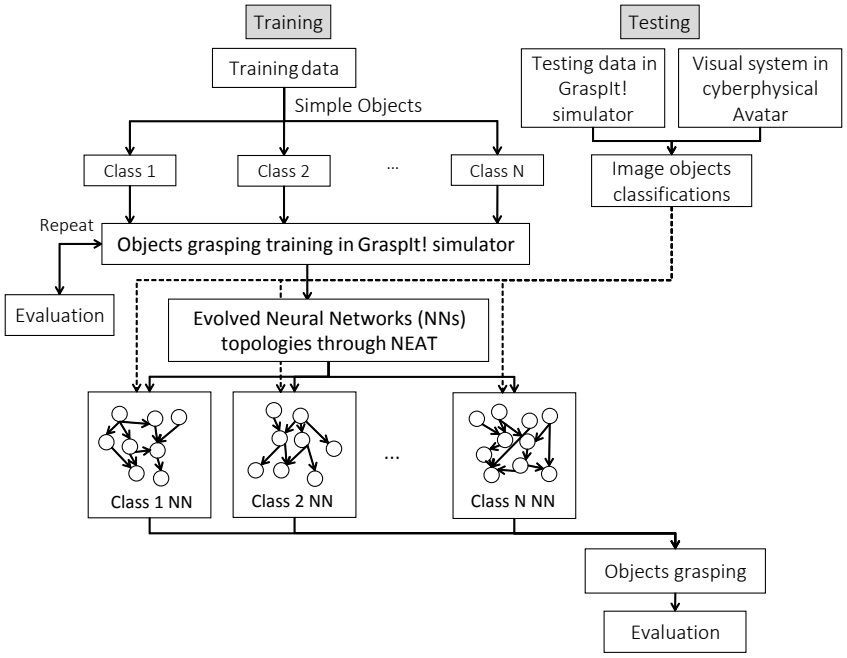


Figure 6.2: The flowchart of the training process and the testing process for the experiments. In the training process, a set of objects are grouped into N separate classes, and then each class produces a neural network through NEAT; in the testing process, the best neural networks can be applied in simulations and tested in a real scenario. The grasping accuracy can be further improved by preprocessing the data before conducting training/testing experiments. These processes can examine if the proposed approach can work.

6.1.3 Testing Combinations of Fitness Function Components

Training experiments are performed with four target objects plus a dining table to vary the scene distribution as shown in Figure 6.3(a), (b). The goal is to gauge which combination of fitness function components (from Section 4.2.2.1) will yield the best performance. Figure 6.3(c) shows the five results of the fitness function for four scenarios through iterative training experiments.

Because grasp quality (f_1) is the most important performance metric, each case must contain f_1, so the combination of total cases is $C_3^3 + C_2^3 + C_1^3 = 7$. The following notation is used to refer to the section: $\{\ \overline{Fit_i} \mid i \in \{12, 13, 14, 123, 124, 134, 1234\}\ \}$.

As an example, $\overline{Fit_{134}}$ denotes the case with f_1, f_3 and f_4. The simulation environment performs a series of simulated grasps on one object on a dinner table for grasping evaluation. Figures 6.4(a)-(d) show training results for grasping a single cylinder, cube, sphere, and mug, respectively. For the cylinder, cube, and mug, the maximum grasping quality f_1 is achieved through $\overline{Fit_{1234}}$ (i.e. each fitness component is helpful). However, Figure 6.4(c) shows that the maximum grasping quality f_1 for a sphere is achieved through $\overline{Fit_{134}}$, which suggests that f_2 does not contribute to better performance. Because the sphere is relatively small, placed in-between other objects it is sometimes blocked by other objects. Because its color is similar to the table, it is hard to distinguish it from the other objects. As a result, NEAT will be mislead by the simple f_2 distance metric.

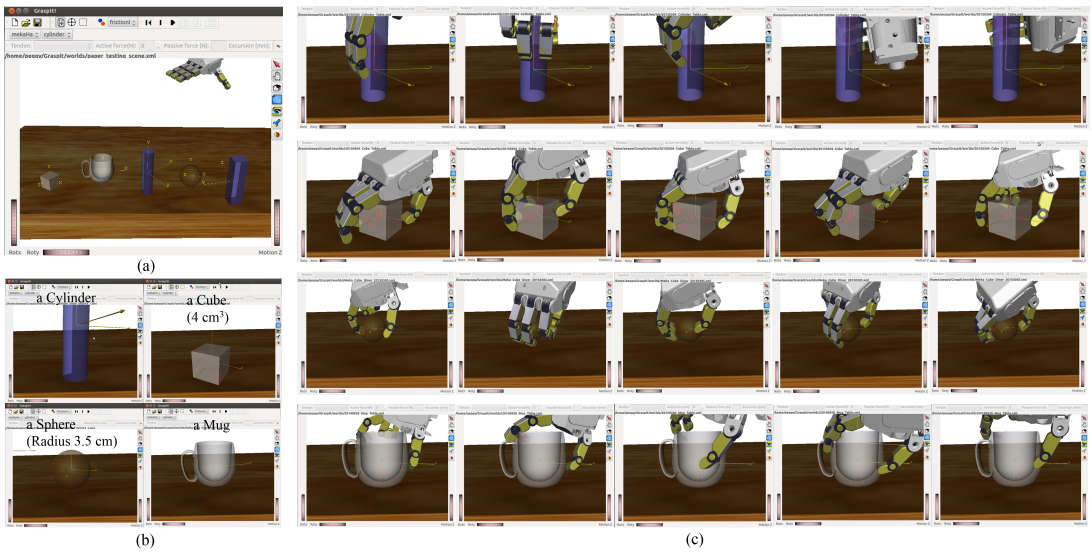


Figure 6.3: Experimental scenarios. (a) A single cylinder, cube, sphere, mug, and cuboid with a dining table and the *Mekahand*. (b) Focus on a single target object each time. (c) The five results for each object during training. The conclusion is that the fitness function can guide *Mekahand* to grasp four different objects.

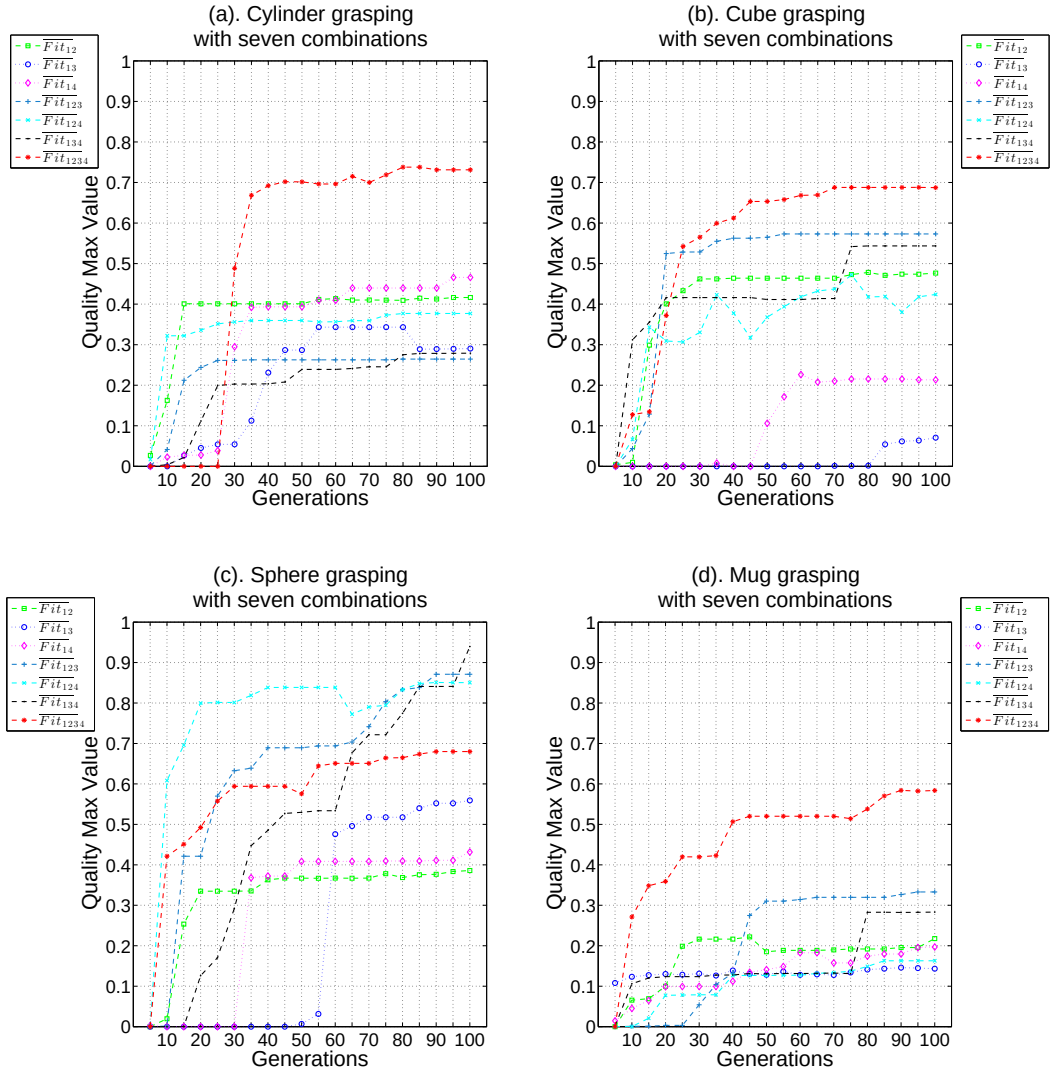


Figure 6.4: Training performance with combinations of fitness components. The training scenario includes a cylinder, a cube, a sphere and a mug, on a dinner table, but the depth sensor focuses only on a single object for each experiment. The x axis represents the number of generations while the y axis represents the normalized grasping quality. These figures show how grasping quality increases over the course of evolution. To evaluate whether each of the four fitness component helps improve performance, (a)-(d) compare seven combinations of fitness components: (Continued on the following page.)

Figure 6.4: (a) shows the results for grasping the cylinder, (b) for the cube, (c) for the sphere, and (d) for the mug. The conclusion is that $\overline{Fit_{1234}}$ produces the best grasping quality for (a), (b), (d), while $\overline{Fit_{134}}$ provides the best one for (c).

6.1.4 Bounding Box Experiments

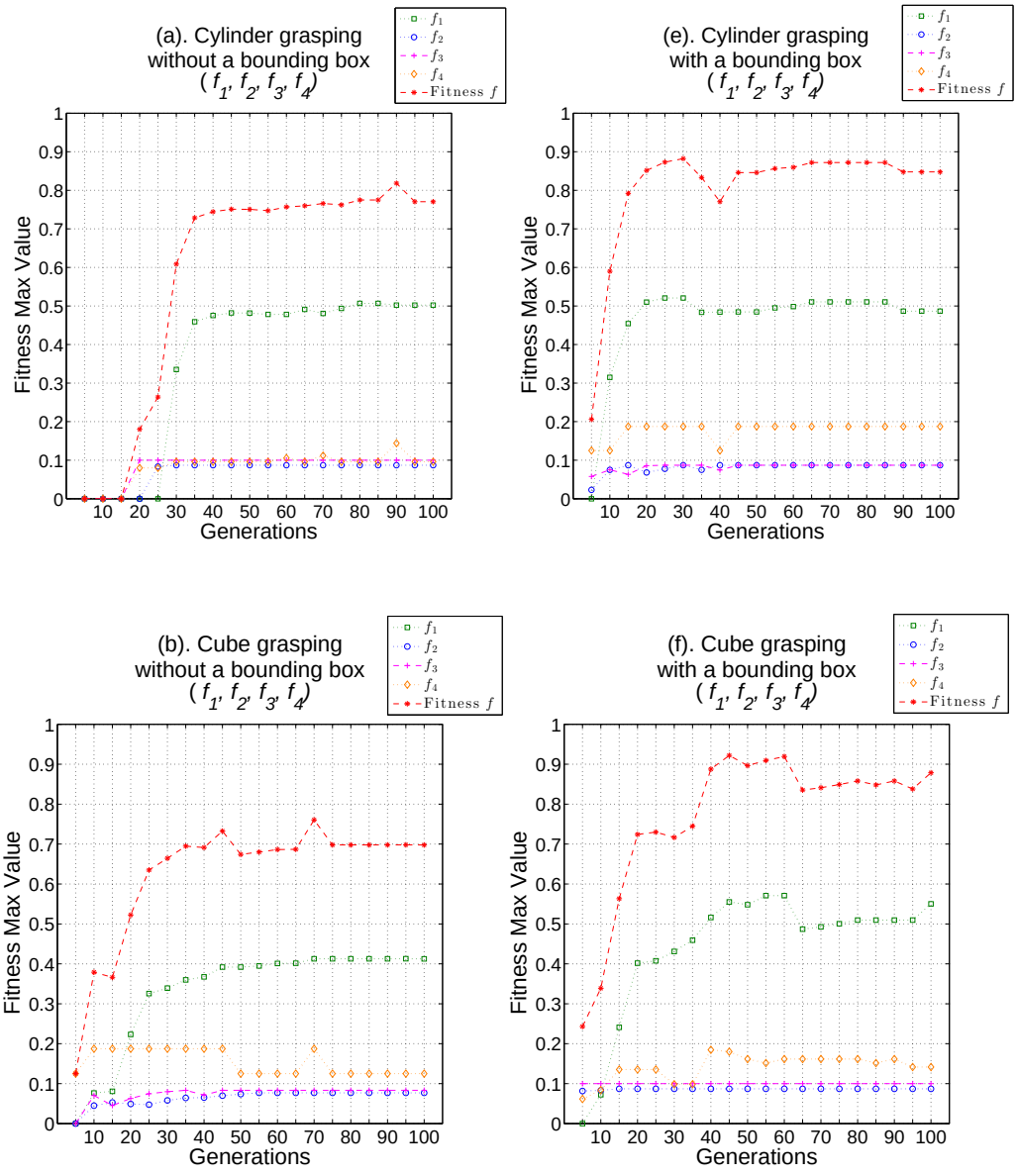
In this section, the approach is applied to a human-supplied bounding box to focus the robots visual processing on the target object, thereby reducing its dimensionality.

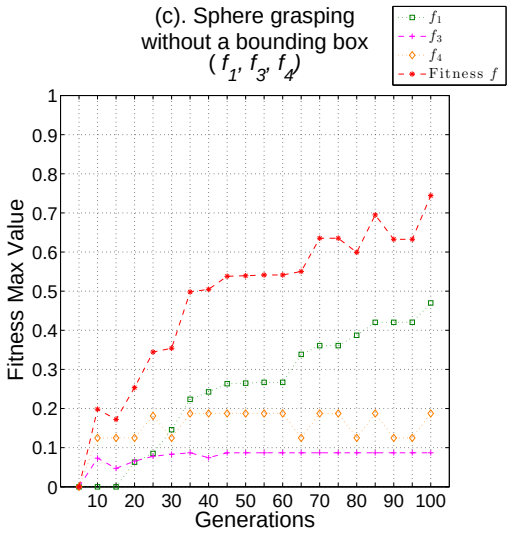
In the second set of experiments, four different training scenarios (without a bounding box) are performed with different target objects, similar to Section 6.1.3. Figure 6.5(a)-(d) show training results for networks trained to grasp a single cylinder, a single cube, a single sphere and a single mug. These figures show how fitness values increase over the course of evolution. Note that larger fitness value implies better grasping quality; also, to differentiate the contributions of f_1,f_2, f_3 and f_4, each of these terms is normalized.

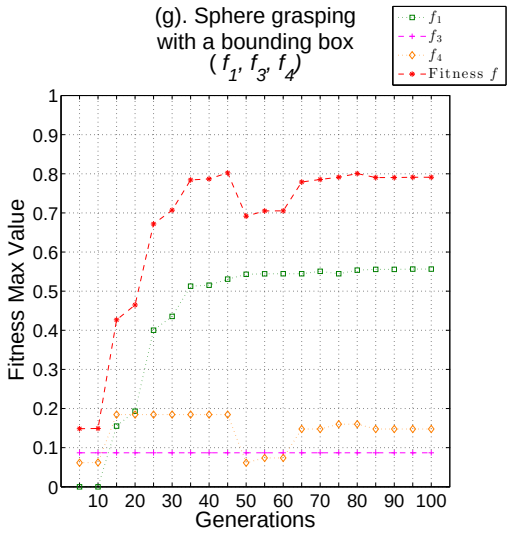
According to the best combination of the four fitness components from Figure 6.4(a)-(d), Figures 6.5(a), (b), (d) differentiate the contributions of f_1, f_2, f_3, and f_4, and Figure 6.5(c) differentiates the contributions of f_1, f_3 and f_4. Note that the maximum score f_1 can attain is 0.6, the maximum for both f_2 and f_3 is 0.1, and the maximum for f_4 is 0.2. Because f_2 and f_3 encourage approaching objects and avoiding obstacles, and f_4 rewards orienting the palm toward objects, that can serve as secondary objectives. These terms are therefore given lower weights than f_1, which measures the grasping quality itself

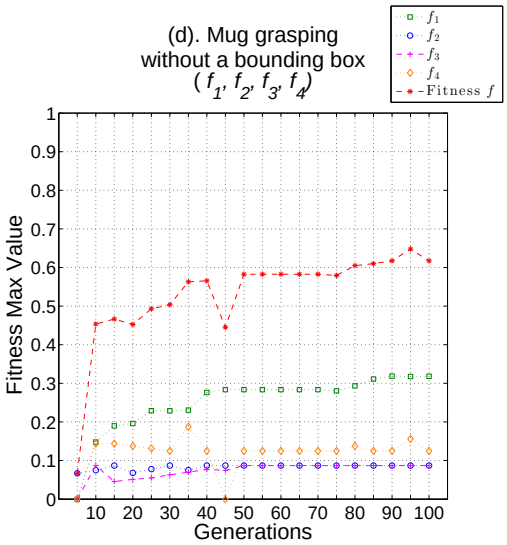
and is thus the most important performance metric.

Because in practice only the best controller would be used, overall best-case results are presented here. To start evolution, individuals in the population are initialized with random weights and a simple topology (i.e. input nodes fully connected to one hidden node, and this hidden node fully connected to the outputs). Because randomly generated policies generally do not cause the robot hand to approach the target objects, low fitness scores are expected. In this stage, f_1 for all the networks is low, so the fitness scores of the networks are mainly determined by f_2 and f_3. These two terms guide evolution to produce networks that approach the objects without being blocked by obstacles. The f_4 component leads the *Mekahand* to the right orientation toward the object. In accordance with this explanation, Figure 6.5(a) shows that initially f_1 is smaller than f_2 and f_3. However, after 25 generations, f_1 becomes dominant. Then, after 90 generations, f_1 reaches its maximum value of 0.5, which means the *Mekahand* can grasp the object more accurately with proper position and orientation. Similar results appear in the other three experiments (Figure 6.5(b)-(d)). In Figure 6.5(b), after approximately 15 generations, f_1 sharply increases, and the total fitness value steadily increases to reach a maximum value of 0.7. In Figure 6.5(c), only f_1, f_3, f_4 are considered, but the fitness value remains around 0.7. In Figure 6.5(d), the fitness value only achieves 0.6. The reason is that it is difficult for the neural network to distinguish the mug object from the other objects. Comparing the four figures, it can be seen that the fitness scores of neural networks trained on the simple


(a). Cylinder grasping without a bounding box (f1, f2, f3, f4)
(e). Cylinder grasping with a bounding box (f1, f2, f3, f4)
(b). Cube grasping without a bounding box (f1, f2, f3, f4)
(f). Cube grasping with a bounding box (f1, f2, f3, f4)
f1
f2
f3
f4
Fitness f
Fitness Max Value
Generations


(c). Sphere grasping
without a bounding box
(f_1, f_3, f_4)
f_1
f_3
f_4
Fitness f
Fitness Max Value
Generations


(g). Sphere grasping
with a bounding box
(f_1, f_3, f_4)
f_1
f_3
f_4
Fitness f
Fitness Max Value
Generations


(d). Mug grasping
without a bounding box
(f_1, f_2, f_3, f_4)
f_1
f_2
f_3
f_4
Fitness f
Fitness Max Value
Generations

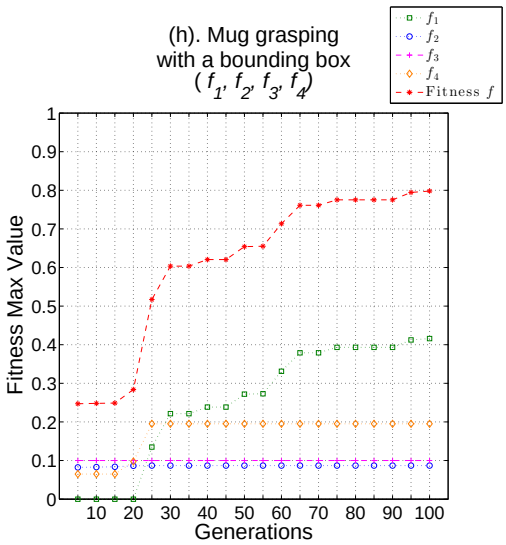

(h). Mug grasping
with a bounding box
(f_1, f_2, f_3, f_4)
f_1
f_2
f_3
f_4
Fitness f
Fitness Max Value
Generations

Figure 6.5: Training performance with and without a bounding box. How fitness values increase over generations is shown for each experiment. A scenario includes a cylinder, a cube, a sphere, a mug and a dinner table, but the sensor only focuses on one single object each experiment. Plots (a) and (e) show a scenario with a single cylinder on a table, (b) and (f) a single cube on a table, (c) and (g) a single sphere on a table, (d) and (h) a single mug on a table. To evaluate whether a bounding box benefits performance, (a)-(d) have no bounding box, while (e)-(h) include the bounding box technique. The total fitness value is shown, as are the contributions from the three or four underlying normalized terms. The conclusion is that the bounding box increases performance, and all experiments eventually evolve ANNs able to grasp the objects in simulation.

objects (Figure 6.5(a)-(c)) were larger than those trained on the more complicated one (Figure 6.5(d)). However, even in the more complicated scenario the networks all learned to approach the target objects and grasp them.

The third set of experiments tests evolution in the same four scenarios, but adds a visual bounding box that can focus the ANN on the most relevant information. The first experiment is shown in Figure 6.5(e). The fitness value gradually increases, and after 5 generations, the values are better than Figure 6.5(a), achieving a value of 0.9 after 45 generations. Also, the fitness values are more stable and do not oscillate around the maximum value as they do in the experiments without a bounding box. Similar results are seen in Figure 6.5(f)-(h). In Figure 6.5(f), the maximum fitness value is 0.92. Figure 6.5(h) illustrates that with a bounding box, more complex object configurations can still produce consistent results around 0.8; although the trend is comparatively slow, the maximum fitness value still approaches 0.7.

This experiments suggest that the more complex the training scenario (i.e. the number of different kinds of objects in the scene), the more difficult it is to train the neural network. Furthermore, if a facet is obscured or the depth array values of an object are similar to the background, then even if the object to be grasped is simple, the training results are poor. However, applying the bounding box significantly improves the results in such cases.

6.1.5 Validating the Generality of Evolved Neural Networks

The training methodology results in neural networks evolved to grasp objects in simulation. To validate such networks, they were further tested in a variety of novel situations through GraspIt! (i.e. situating for which object was not explicitly trained). Most objects in the scenes were not seen at all during evolution or not placed in the same location, and their arrangement is new. The experiment thus measures how general the evolved solutions are. A successful case is recorded if the *Mekahand* can grasp the object; otherwise it is recorded as a failure.

For this generality test, each object was tested 100 times. The grasping procedures were implemented under test conditions randomly placing the different sizes and textures of a cylinder, a cube, a sphere, and a mug, as shown in Figure 6.6, at different positions and orientations on the table. The evolved neural networks in 6.1.4 were labeled as Cylinder (N_A), Cube (N_B), Sphere (N_C), and Mug (N_D), and based on similar classification of objects, the most appropriate neural network was chosen for testing. The success rate

in Table 6.1 compares the neural networks with the different objects. These results show that despite its simplicity, the proposed bounding box method still performs reasonably well in grasping novel objects. However, if the target object is too far from the center of the image frame, the neural networks often perform unreliably, indicating the training process may need further refinement to deal with such boundary cases. Table 6.1 shows the best results from among all the experiments. Also, in some cases the *Mekahand* collides with objects while grasping, because many objects are placed on the table. A potential remedy is to decompose the movement into more steps to avoid such collisions. One way to do so would be to rely on additional input from the human supervisor.

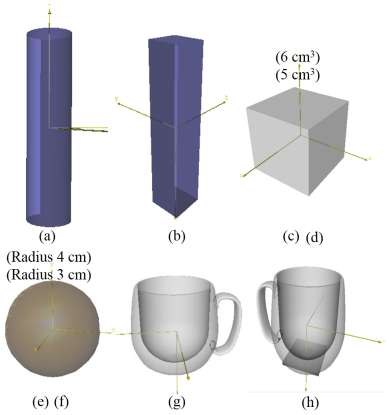


Figure 6.6: Testing different sizes and textures of objects across novel locations and orientation. Shown in the figure are a cylinder, a cuboid, a cube, a sphere, a mug and a plated mug. Note that the letters labeling each object correspond to similar labels in Table 6.1.

Next, Subsection 6.2 presents transferred to the real robot.

Table 6.1: Generalization results of grasping objects at novel positions with evolved networks. The results with a bounding box outperform the ones without a bounding box, which indicates that a bounding box is an effective way of increasing grasping performance.

#	**Objects**	**Class**	**without a Bounding Box**	**with a Bounding Box**
(a)	cylinder	N_A	52%	89%
(b)	cuboid	N_A	65%	81%
(c)	cube (6 cm^3)	N_B	69%	76%
(d)	cube (5 cm^3)	N_B	73%	82%
(e)	sphere (radius 4 cm)	N_C	71%	88%
(f)	sphere (radius 3 cm)	N_C	68%	80%
(g)	mug	N_D	71%	85%
(h)	plated mug	N_D	62%	74%
	Mean/Std		66.38% (±6.80%)	81.88% (±5.33%)

6.2 Validating with Dreamer

Beyond simulated results, learned policies were also transferred to the physical world. A physical (i.e. not simulated) Kinect sensor was applied to capture object depth array information. This information was provided as input to an evolved neural network to guide *Dreamer* robot's grasp.

6.2.1 Kinect Sensor Implementation

In order to retrieve the Kinect sensor data and feed it into the system, the sample program regview provided by OpenKinect project[2] was modified.

[2] `http://openkinect.org`

This program was enhanced to be run as a server that waits for the connection from the remote-control PC over the TCP/IP. Besides, it was tweaked to register the video format as FREENECT_DEPTH_REGISTERED. The reason is that in the Kinect sensor, the depth camera and the color camera are two separate sensors, which means their views are different. Only by doing so, the depth data will be projected to the view of the color camera. In this video mode, the depth data is in millimeters, and the pixel coordinates can be translated from (i, j, z) to (x, y, z) as follows:

$$\begin{aligned} x &= (i - \text{width}/2) * (z + \text{minDistance}) \\ &\quad *\text{scaleFactor} * (\text{width}/\text{height}), \qquad (6.1) \\ y &= (j - \text{height}/2) * (z + \text{minDistance}) * \text{scaleFactor}, \qquad (6.2) \\ z &= z, \qquad (6.3) \end{aligned}$$

where minDistance = -100, scaleFactor = .0021 and weight and height are the size of the image. The x, y, z is a right-handed Cartesian system: with z axis perpendicular to the Kinect image towards the image, x-axis points to the left, and y-axis points up. Before sending commands to the robot the coordinates are transformed again to match *Dreamer*'s coordinates.

6.2.2 Remote Control Panel

Figure 6.7 shows a screen capture of the remote-control application for supervising *Dreamer*. The remote-control user interface shows the color images and depth images from the Kinect camera; the images from the IP camera are

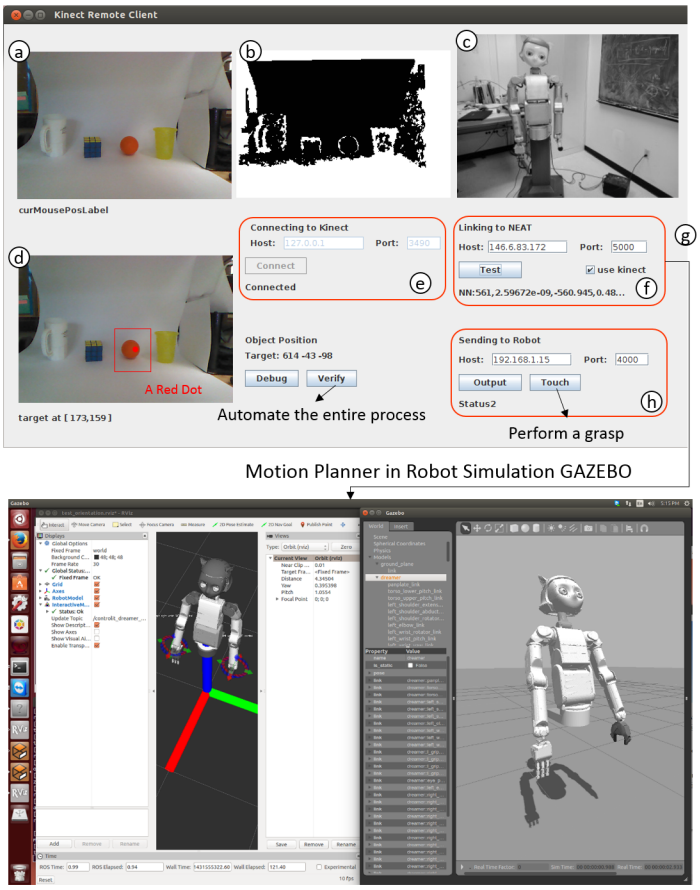


Figure 6.7: A screen capture of the remote-control software application for supervising the *Dreamer* robot. (a) Color and (b) depth images from the Kinect sensor. (c) The image from the IP camera. (d) An image snapshot taken when the user clicks on the color image. (e) A dialog for connecting to *Dreamer* through a computer network. (f) A dialog for inputting the captured depth array into an evolved ANN. (g) Use motion planner to obtain a trajectory. (h) A dialog for sending the orientations and positions from the ANN to *Dreamer* to control its grasp. The conclusion is that the grasping experiment can be implemented through the remote control panel.

displayed in the third image panel at the top of the user interface.

To automate the high-level supervision of the grasping experiment, six commands was implemented on the remote control panel: Connect, Test, Output, Touch, Verify, and Debug, as shown in Figure 6.7. When the supervisor clicks on the Connect button, the computer connects to the Kinect sensor to capture depth information. Then, when the Test button is pressed, the depth array is provided to the evolved neural network as an input. After executing the neural network, its outputs are interpreted as coordinates and orientation of the hand for grasping the object. When the Output button is pressed, the results are sent to *Dreamer* and the robot is directed to approach the object. Finally, when the Touch button is pressed, *Dreamer* will grasp the object using the grasping information provided by the neural network. After *Dreamer* obtains this information, i.e. the grip orientation and position, the controller PC computes the distance between the *Mekahand* and the object, predicts the hand's trajectory, and approaches the object. Once the *Mekahand* is near the target object, the thumb and the three finger motors are synchronized to perform the grasp. A Verify button is provided to automate the entire process for convenience; the Debug button serves to aid in system debugging, providing coding logging information.

6.2.3 Transitioning to Physical Controller

An automated grasping platform was built to demonstrate this process. The networks evolved in simulation are transferred to this platform to evalu-

ate them in a physical environment. To carry out an experiment, a human experimenter uses the control panel to choose a target either without or with a bounding box in the color image from the laptop screen with the Kinect sensor by clicking on it. After designating the target, a copy of the color image is copied to the target object panel, and a red dot is added on the image, indicating the position of the click. The depth data at that point is used to calculate the approximate position of the object to be grasped. This results specifies the grasping task for the robot to perform. Note that the grasping behavior was not evolved on the actual robot, but was transferred from simulation.

The video[3] demonstrates [30] grasping of novel objects from the simulation to the real *Dreamer* robot. Besides, Figure 6.8 shows screen captures taken from a proof-of-concept demonstration of grasping a tennis ball, a bottle, ball, a Rubik's cube and a cup. *Dreamer* can successfully approach and grasp target objects when controlled by an evolved neural network. Since these objects were not seen during evolution, the experiment demonstrates two achievements: learning transfers from simulation to the real world, and it generalizes to grasp objects. Quantifying how well grasping works still needs a metric for the assessment of the quality of a real grasp, so further work to incorporate real sensor data on the *Mekahand* (e.g. touch pressure) is ongoing.

[3] `http://www.cs.utexas.edu/~peggy/rtss2015.html`

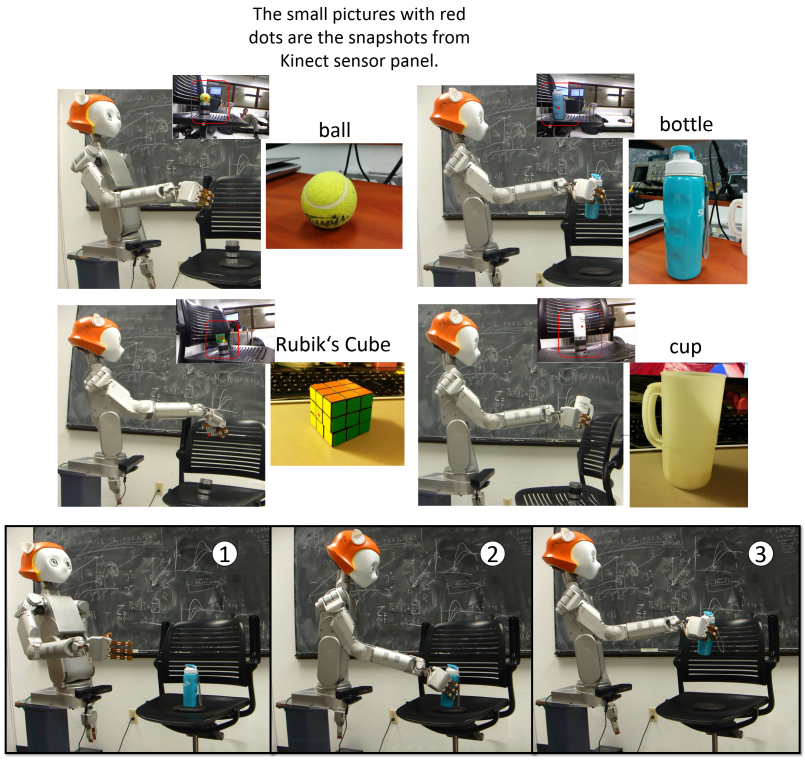


Figure 6.8: Screen captures from the video demonstrating *Dreamer* grasping a ball, a bottle, a cube and a cup through an evolved controller. Note that the small picture with red dots are the snapshots from Kinect sensor panel. The bottom snapshots labeled with (1)-(3) represent the object grasping process, from the initial, approach to grasp a bottle. The figures confirm that transferring results from simulation to reality is possible, and applying the approach generalizes to novel objects.

The video link is `http://www.cs.utexas.edu/~peggy/rtss2015.html`

Chapter 7

Case Study: Autonomous Pick-and-Place System

To enhance the functionality of the semi-autonomous robotic system in cyber-physical applications, we built an autonomous pick-and-place robotic system upon the semi-autonomous robotic system. This chapter presents the architecture of the autonomous pick-and-place system and its flowchart in Section 7.1. The two major fundamental elements in this system are the perception module presented in Section 7.2 and the manipulation module described in Section 7.3. Finally, Section 7.4 presents the environmental setting, and hardware/software system overview, and implementation, and experimental results for the pick-and-place task.

7.1 The Overview of the Robot System

Having described these techniques in Chapter 4 and how we design the actual architecture of cyberphysical avatar, we turn to present a case study: the autonomous pick-and-place robotic system.

The pick-and-place system comprises the robotic arm *Hoppy* with its

control system developed by *Robotics Robotics* (*RR*) Ltd. [69], and its end-effector with the specialized gripper with suction function, and a cost-effective, real-time and 3D vision system devised computer vision algorithms, including object recognition and object localization, and 3D model reconstructions, and a real-time physical network. The specific task explored in this case study is to recognize and instruct the robotic arm to pick up different types of objects and place into a box in the real-time environment. The physical realization of the pick-and-place system has been implemented in the Cyber-Physical System (CPS) Laboratory at GDC [1].

Figure 7.1 illustrates the architecture of the pick-and-place system. *Hoppy* consists of a six-DOFs robotic arm, a desktop PC (*controller PC*) exercising Ubuntu 14.04 with the RTAI patched real-time Linux kernel that executes the models and control infrastructure to govern *Hoppy*'s behavior and the other *vision PC* is built to run on top of Ubuntu 14.04 distribution with Linux kernel version 3.13.0 that implements machine learning and vision algorithms. The robotic control framework is based on ROS (Robotic Operating System) indigo with the Gazebo simulation[1] 2.0 and 6.0. Two types of cameras are installed in the system. An Xtion Pro sensor is connected to the *vision PC* and is installed in front of the gripper to capture color images, depth information and generate 3D point cloud data, and the other camera is installed in the back of the robotic arm to capture *Hoppy*'s surrounding environment. The *vision PC* communicates with the *controller PC* through ROS

[1]Gazebo, `http://www.gazebosim.org/`

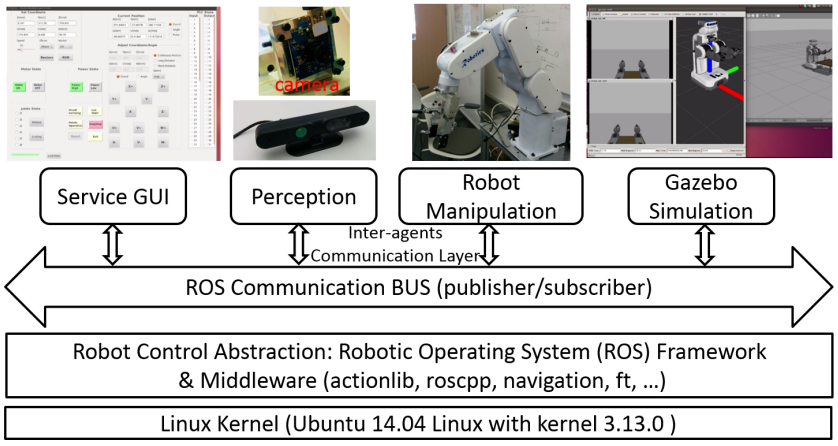


Figure 7.1: The architecture of the pick-and-place system. All the messages are communicated and transferred through the ROS topic system upon Ubuntu Linux 14.04. At a high-level layer, the autonomy of the system is governed by a designed service graphical user interface (GUI) which controls perception or robotic manipulation, and the simulation environment Gazebo can be simulated the robotic action before transferred to the real world. A service GUI acting as the brain of the system decides which actions can be executed based on the sensors and control feedback.

communication bus provided by a ROS master server; the *vision PC* yields 6D poses to the *controller PC*, and the *controller PC* plans the coordinates and position of the desired trajectory to reach the object and move the robotic arm *Hoppy* to grasp it.

The two main modules in the proposed system are the perception and manipulation. The perception module is used to run learning and vision algorithms, whereas the robot manipulation is used to control *Hoppy*'s arms. To realize the algorithms for *Hoppy*, the 3D model of *Hoppy* with a proportional-integral-derivative controller (PID controller) was designed and implemented

in an open source simulator Gazebo, and encapsulated with ROS format to interface with the Gazebo simulation to simulate the trajectory planning and object grasping in realistic scenarios. Besides, we also provide a service graphical user interface (GUI) for a supervisor to issue the high-level commands for the robotic hand to perform the pick-and-place task as well as monitor the system status. All the messages are communicated and exchanged through the ROS topic system which is a pub-sub like a system that can efficiently multiplex numerous messages between different modules and thus decouple the modules such that each module can be cleanly implemented and improved to achieve the reusability and maintainability.

Figure 7.2 depicts the flowchart of one pick-and-place experiment. The experiment is achieved as follows. First, a service GUI keeps polling the status from either the perception module or the robotic manipulation module. If a human supervisor adds a grasping task by the GUI, the perception module will start consuming the video streaming input from camera sensor and kick off the pipeline to deal with object recognition and propose an appropriate pose for grasping the desired object. Further, the pipeline can be broken down into four steps, including (1) reading video streams from the depth sensors, (2) predicting and marking the Region-Of-Interest (ROI) of an object with its label, location, and probability, (3) estimating the best pose to pick up the object by using LineMOD [26] and 3D models, and (4) transforming the object pose into the robotic arm's poses. On the other hand, the manipulation module is decomposed into five steps, including (1) performing trajectory planning,

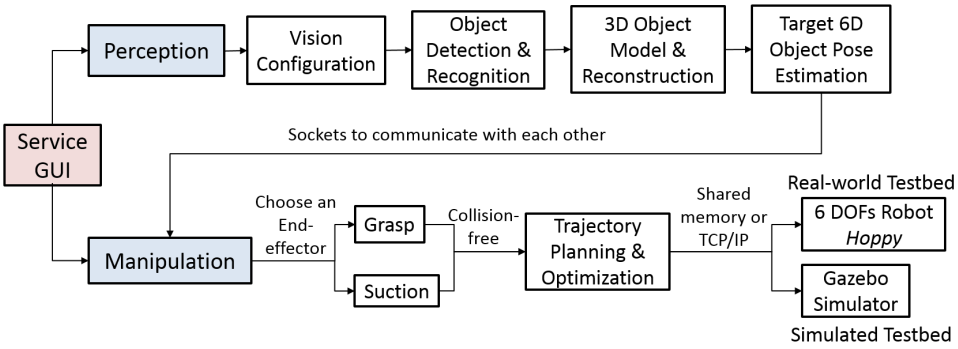


Figure 7.2: The flowchart of the pick-and-place system. A service GUI for the supervisor to issue the high-level commands to perform the pick-and-place tasks and monitor the system status. The functionalities of perception module are to do the object recognition and to suggest the best poses for object grasp/suction; the tasks of manipulation is to take the poses and object position output from the perception part, and then to perform trajectory planning and controls/moves the physical robotic hand. We adopt the Gazebo simulation for the simulation and then applied for *Hoppy* robot.

(2) moving the physical robotic hand to approach to the object, (3) grasping/suctioning the object from an original position to the requested position, and (4) releasing the object. Before we actually apply and operate our system in a real-world experiment, (5) we will try them in the Gazebo simulation as well.

Next, we describe each module in more detail.

7.2 Perception Methodology

In the pick-and-place robotic system, a fast and accurate perception module is essential, allowing us to detect and recognize the target items from a shelf without specialized sensors, and further yield an estimation from a

set of feasible grasp candidates. Figure 7.3 illustrates the procedure of perception module. In vision configuration, four NVIDIA Tesla GPU K80 cards under Ubuntu 14.04 are used for accelerating images training process. We also installed CUDA driver[2], CUDA 7.5 SDK toolkit[3], the Xtion driver running by using OpenNI2 SDK[4], and the camera driver. For object detection and recognition, we investigated that image pre-processing can improve the capability of image features extraction and further characterize each type of objects. Then, we use these features to train a model through convolution neural networks (CNNs). For 3D object model and reconstruction, we construct every 3D model of the object and built a database (DB) to convert a set of objects information and then store in a DB. For target 6D object poses estimation, we use LineMOD [26] to implement template matching and applied optimization algorithm for the point clouds data to improve 6D pose estimation. More details are described in the following subsections.

7.2.1 Object Detection, Recognition and its Region-of-Interest

For images pixels, the model can be represented by a region-based convolutional neural network (R-CNN) architecture. R-CNNs have great advances in computer vision and usually focus on object detection and image classification. The architecture uses region proposal network [20] built by several fully connected CNNs to generate potential bounding boxes from an

[2]Nvidia CUDA driver, `http://www.nvidia.com/`
[3]Nvidia CUDA toolkit, `https://developer.nvidia.com/cuda-toolkit-archive`
[4]OpenNI2, `https://structure.io/openni`

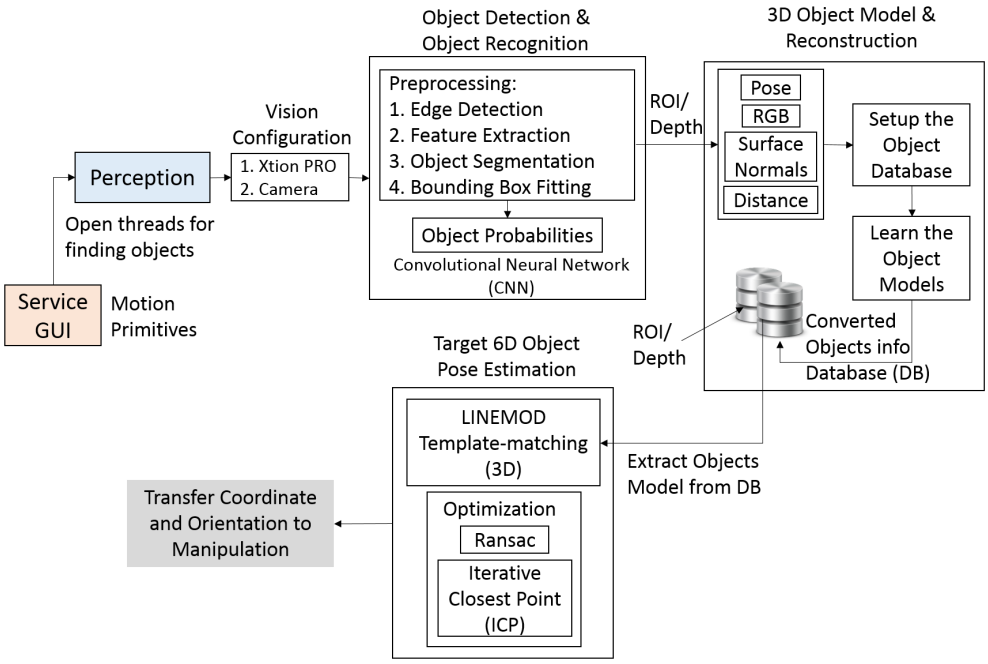


Figure 7.3: The flowchart of the perception module in the pick-and-place system. In the procedure of this module, a camera and CUDA driver and toolkits are installed for vision configuration; raw images are preprocessing and calculated each object's probability for object detection and recognition; a database (DB) was constructed to save objects information for easily fetching for 3D object models and reconstruction; LineMOD [26] was utilized to execute template matching and optimized coordinate and orientation for target 6D object pose estimation. Finally, 6D pose estimation was sent to the robot manipulation module. The conclusion is that the system integrates real-time vision, 3D reconstruction, and template matching to predict object's orientation and coordinates.

image. To generate bounding boxes, a selective searching method with sliding windows is applied [8] and each sliding window is a small network to slide over the conventional features map. In [21], *max-pooling layers* can reduce dimensions of representations by performing a nonlinear downsampling operation, which is useful to reduce computation for upper layers and provide a form of translation invariance. *Region-of-Interest pooling layer* can adaptively adjust max pooling layer and dynamically expand batch size. *Fully-connected layer* can compute the likelihood of classes. *Bounding-box regression* is performed on features pooled from ROIs, and the regression weights are shared by all region sizes.

The current detection and recognition system also adopts a sliding window approach where each object is classified and evaluated at different locations over the entire image by using a classifier [14]. More recent approaches like a *region-based convolutional neural network* (R-CNN) architecture, Girshick et al., [20] proposed a region proposal method which is to use region proposal networks (RPNs) to feed to R-CNN as an input and then train a classifier from the proposed bounding boxes of each object, called *Region-of-Interest* (ROI), but the computation is time-consuming and difficult to converge because each component must be trained individually. More performance improvement such as fast R-CNN and faster R-CNN can be seen in [19, 65] that train multiple objects for a single convolutional network simultaneously. The similar idea is from Redmon et al., [64], that simplified to a single regression problem and introduced *You Only Look Once* (YOLO) architecture which

can predict each class probability and the coordinate of bounding box from extracted features of images.

According to our observation, several difficulties are yet required to be conquered, such as semi-transparent objects, varied ambient brightness, objects reflections inside the shelf, and the bad angles of view because of the narrow shelf. To overcome these hurdles while satisfying the real-time detection constraints, we adopt a deep neural network based on YOLO [64] which can quickly detect, scale and locate each object. The detailed experimental setup and results for the training and the testing process are shown next.

7.2.1.1 Training Process

First, we create the dataset that contains about 40,000 pictures (there are 18 examples objects from Amazon Picking Challenge competition (APC) and each object has different 2,500 images shots.) of 18 objects in different orientations and with random backgrounds, and each image is labeled and annotated with ground-truth locations of rectangles. Figure 7.4 shows that the dataset covers a large variety of categories. Second, we implement the scripts to semi-automatically create the annotation which denotes object type and coordinate of a bounding box for each object within each image which are further converted to YOLO input format and merged to form a training list. Third, the varieties of objects detection are unified into train a single convolutional neural network altogether with the extracted features of entire images and the bounding boxes, and the trained network model can predict the

bounding boxes for these objects and their classes likelihood simultaneously.

Because the raw data from the Xtion sensor is 640×480, according to Redmon et al., [64], the pixel array was subsampled to form a reduced 448×448, and each image of bounding box was divided into an 7×7 grid cell. We have tried a larger cell, such as 14×14, however, the training process was so time-consuming that we decided to keep 7×7 as default cell size. This grid cell is in charge of the detection of that object when its center falls into one of grid cell region. The class prediction probabilities can be defined as $P_r(Class_i) * IOU_{predict}^{truth}$ [64], where $P_r(Class_i)$ represents the probability of this class, and $IOU_{predict}^{truth}$ represents the intersection over union (IOU) between the predicted box and the ground truth, and this pivot provides us a score for each box to predict which class appears in this box and how good for the predicted bounding box. For training our model, we also use the same parameters, 7×7 grid cell, and has 18 different labeled classes, so our final prediction is a $7 \times 7 \times (5 \times 2 + 18)$ tensor.

The network architecture expects to acquire the class probabilities and coordinates by means of transformation images to extract feature points through CNN layers, pooling layers, and fully-connected layers. The architecture of the neural network is illustrated in Figure 7.5, including an input layer, followed by 9 convolutional layers and 5 maxpool layer alternately, followed by 3 fully connected layers, and a dropout layer and an output layer. Let a volume of size be width($\widehat{W}$) $\times$ height($\widehat{H}$) $\times$ depth($\widehat{D}$), and the number of filters/kernels (called filters size) be $\widehat{K}$, and their spatial extent (called the

Figure 7.4: Construct the dataset of example objects from Amazon Picking Challenge competition (APC): (a) a duck toy, (b) a brush, (c) a tennis ball, (d) a set of cups, (e) a yellow toy, (f) a green toy, (g) a set of balls, (h) wall plugs, (i) pens, (j) a box, (k) cloth, (l) a pencil box, (m) a book, (n) glasses, (o) a bottle of water, (p) a sparking plug. The taken pictures from these objects have been annotated with their bounding boxes and categories. The dataset will be used to train our recognition model.

receptive field of the neuron) be $\widehat{F}$, and the stride be $\widehat{S}$. To refer to previous work [64, 66, 71], the input layer is set by $448 \times 448 \times 3$ ($\widehat{W} \times \widehat{H} \times \widehat{D}$), and each reduction layer is set at $\widehat{F} \times \widehat{F}$, where $\widehat{F}$ is 7, 1, 3, respectively, and followed by the maxpooling layer, and the output layer is set at $7 \times 7 \times (5 \times 2 + 18) = 1372$. In Figure 7.5, the first convolutional layer is 7×7, and has 16 filters (or kernels) of size, and we denote it as 7×7@16. The stride $\widehat{S}$ and the size $\widehat{F}$ are set to 4. After the maxpool layer, the reduction layer is $56 \times 56 \times 16$. Note that, the spatial extent always equals the depth of input layer. Then, we process them in the same way to produce a $7 \times 7 \times 64$ and followed by the two fully-connected layers (256 inputs and 256 outputs, and 256 inputs and 4096 outputs, respectively). To avoid overfitting and combine different neural network architectures more efficiently, the dropout layer was added to temporarily remove the unit out from the neural network such that the neural network is more general and more accurate, and the probability $\widehat{p}$ is set to 0.5. The networks were trained for approximately a week, and we used Darknet framework for all training and inference [2]. The loss function is the summation of square errors for the coordinates, widths, lengths, class probabilities between the predicted bounding boxes and their ground truth, and the penalty is classification error.

Because the trained model is large-scale visual recognition, the results could take months of work if computationally inefficient. To tackle the problem, an open-source deep learning framework Caffe [35] is adopted, and it can not only be an extensible toolkit for state-of-the-art deep learning algorithms

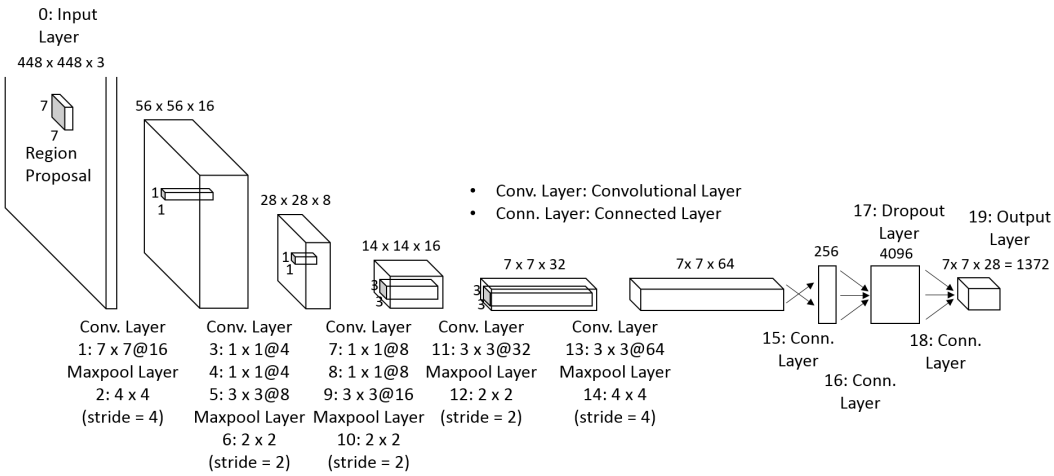


Figure 7.5: The architecture of the neural network for training detection objects. The network has 20 layers, including the input layer, 9 convolutional layers, 5 maxpooling layers, 3 fully connected layers, 1 dropout layer, and the output layer. Based on previous work [64, 66, 71] , the input resolution for classification task is $448 \times 448 \times 3$ (width $\times$ height $\times$ depth), and followed by a series of convolution layers and maxpool layers. Here, the range for the 1^{st} convolutional layer is 7×7 and has 16 filters, denoted as $1 : 7 \times 7@16$. The size in the 2^{nd} maxpool layer is 4, denoted as $2 : 4 \times 4$. The convolutional and maxpool layers interchangeably reduce the features space. Then, followed by the two fully-connected layers, the first is 256 input/output features, and the second is 256 input and 4096 output features. Next, a dropout layer is applied to increase the accuracy (the probability is set to 0.5), and followed by a fully-connected layer. Finally, a $7 \times 7 \times 28$ neural network was produced. The trained network model can be used to predict objects and its coordinates of bounding boxes.

but support CUDA code that can be run on GPU parallelly. Our network was implemented and ran with a NVIDIA Titan GPU with 10 cores, and 4 of them are allocated and fully utilized to conduct the experiment. Our system was compiled with CUDA 7.5 and CUDNN 3, and the batch size setup is 32 for all neural network. With the setup, the training time is greatly shortened by a factor of two.

7.2.1.2 Testing Process

To preserve generality, the position and orientation of the objects are randomized. The experimental parameters are the same as the training setting. Our trained model was tested within $100cm$ for all the objects in any spaces of the shelf, as shown in Figure 7.6. An encompassing bounding box is automatically generated and centered on the desired object, and we also can predict multiple objects at the same time. Impressively, the objects like a bottle of water or a glass, despite their transparent properties, the trained deep R-CNNs model still can correctly predict what objects are present and where they are.

7.2.1.3 3D Object Coordinates

The CNNs through training can detect the ROIs and label objects with probabilities by integrating information from a high-dimensional image provided by the Xtion sensor. Because an encompassing bounding box is automatically generated and centered on the desired object, the boundary

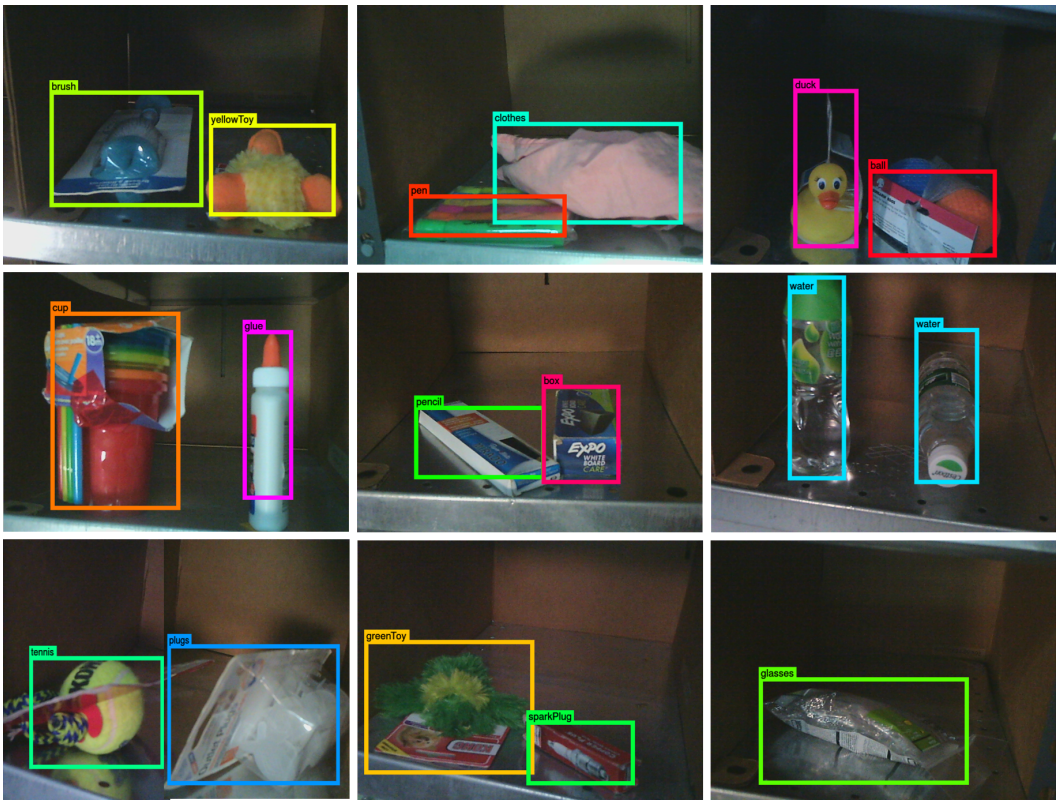


Figure 7.6: The testing results of the region-based convolutional neural networks (R-CNNs) model. An encompassing bounding box is automatically generated and centered on the desired object through the trained CNNs model, and also shows each object name and its width, length and coordinates. The figure shows that the model can precisely detect the object and its locations.

range can be mapped to four coordinates. To find the object of 3D positions, the center point of a 2D plane of each object (x, y) is determined, and (x, y) of each object can be mapped to the normalized depth raw data z. Then, the object 3D location can be obtained.

7.2.2 3D Object Model and Reconstruction

According to the research survey from open source community, we found Willow Garage Company[5], the developer of PR2 robots, started a project, called the Object Recognition Kitchen (ORK) [88] for object recognition. ORK contains input/output handling, robot/ROS integration, and database management. Therefore, a database was first constructed to store all models and the creation of every objects' models with point cloud library (PCL) in 7.2.2.1. PCL [72], an open source C++ library, provides state-of-the-art algorithms for 3D perception and supports point cloud processing and 3D geometry processing, such as filtering, registration, and recognition, etc. Then, a 3D reconstruction of each object was created and mentioned in 7.2.2.2.

7.2.2.1 Acquisition Setup and Procedure

- **Setup:** A database was stored with every object's mesh, ID name, images, the related information, and also should be recorded when/who/where added the objects. Apache CouchDB [17] software was adopted as our database to record objects information because of

[5]Willow Garage Company, `http://www.willowgarage.com/`

usability, simplicity and practicality [63]. CouchDB is an open-source documentation-oriented database management system, and developed by the Apache Software Foundation, in which querying and indexing are performed by using JavaScript language. Then, indoor illumination was controlled in constantly dark through the entire recording process with single light source from the upper front of the objects. Next, every object was placed on top of a board which is supported by a robotic arm that allows us to rotate the board precisely for the requested angles. Then, the Xtion sensor was fixed on top of the room and placed about $50cm$ in the front of an object recorded. Finally, the program enables that each object was rotated by itself per 20 degrees, and was taken around 50 pictures.

- **Procedure:** First, the CouchDB database was installed, configured and run as a service in the background. Then, each object was placed on top of the board and rotated 18 times where each time it is rotated by 20 degrees. Next, we configured the controller to do the rotation for different angles and recorded the entire process so that we can get precise data. The depth/RGB video stream was captured from the Xtion camera and then can be utilized to roughly create 3D reconstructions of each object where the properties and the models of the reconstructed 3D objects are stored in JavaScript Object Notation (JSON[6]) format and

[6]JavaScript Object Notation, `http://www.json.org/`

inserted into the CouchDB database for later use.

7.2.2.2 Generate Objects Model

To generate objects model, the tools, *object_recognition_core* and *object_recognition_capture* [88], we adopted can provide ORK infrastructure to capture and transcribe the real world object into a 3D digitized form. To capture objects, a fiducial marker likely a dot pattern attached to the real world object, was required to obtain accurate pose estimation, stable background segmentation, and a consistent object coordinate. The 3D points capture was evenly distributed in different points of view while the object was centered at the frame and then can be produced a ROS bag of data. Figure 7.7 shows the screen snapshots of the samples of the procedure for 3D reconstruction. In Figure 7.7(a), we perform single camera tracking and reconstructions of 3D objects' shapes, including a duck toy, a box, a brush, a pencil box and a glue. In Figure 7.7(b), the static background can be filtered out to extract only the foreground objects.

Then, *object_recognition_reconstruction* is used to create an approximate untextured mesh from the captured videos, and this step merges the depth images from different points of view. Figure 7.8 visualizes the creation of 3D object meshes which contains a set of balls, a box, a brush, a cup, a duck, a glue, a pencil box and a spark plug; Figure 7.9 depicts the objects information which is stored in the CouchDB database, and the construction of customized database can be used for searching pose estimation of each ob-

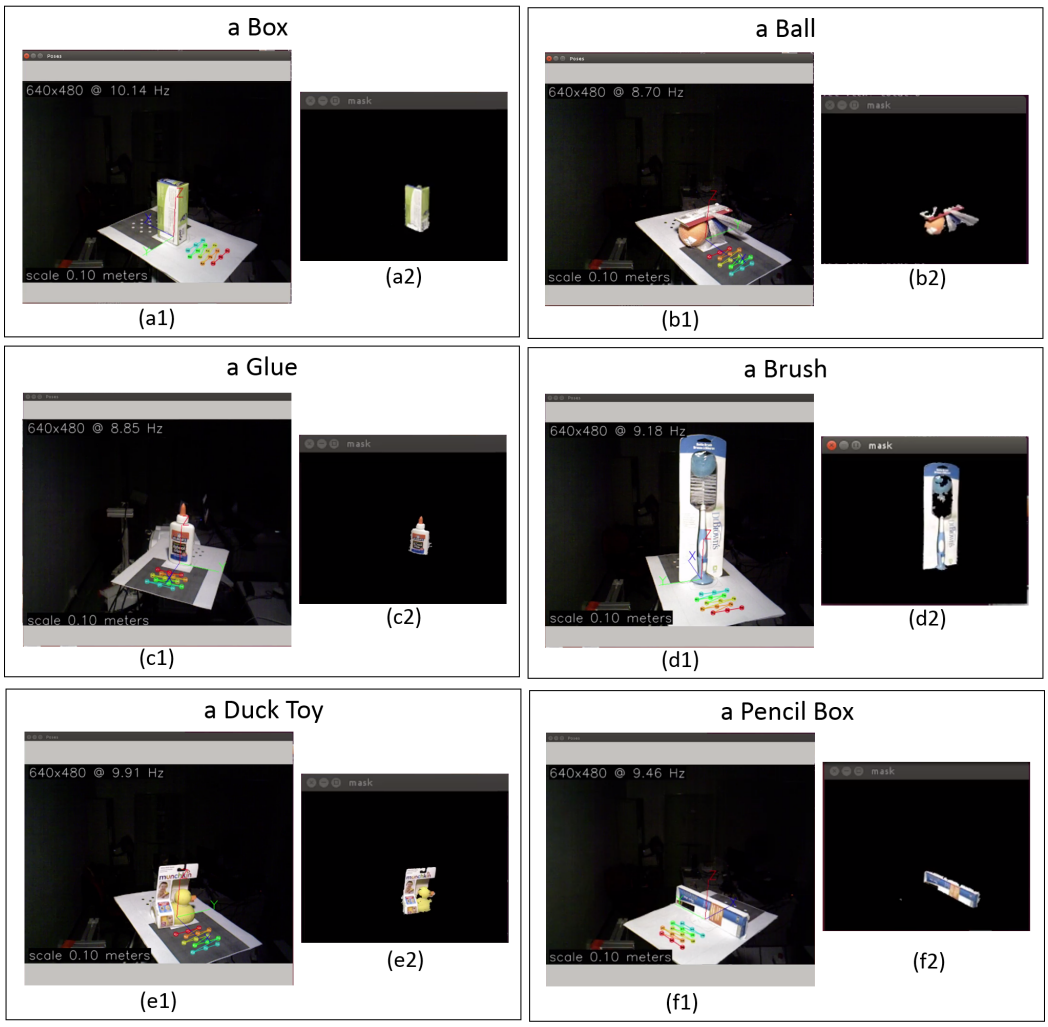


Figure 7.7: The process of 3D objects reconstruction. (a). Screenshots captured from the recorded videos demonstrating a duck toy, a box, a brush, a pencil box and a glue. (b). Screenshots capture from the recorded videos displaying the extracted objects. The conclusion is that this information can be saved in the database to build 3D meshes.

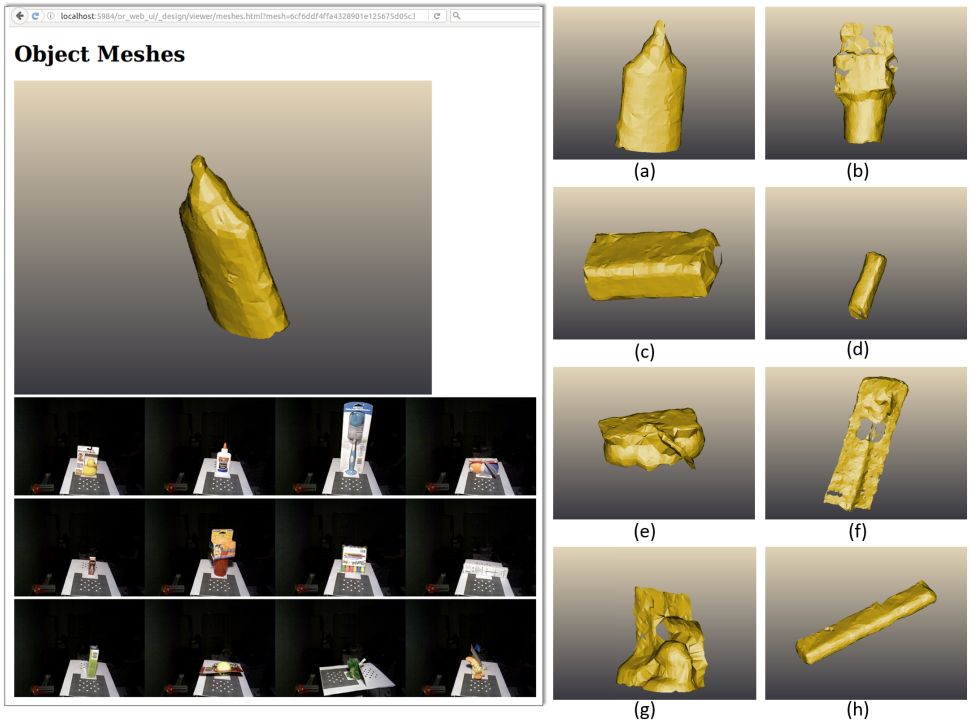
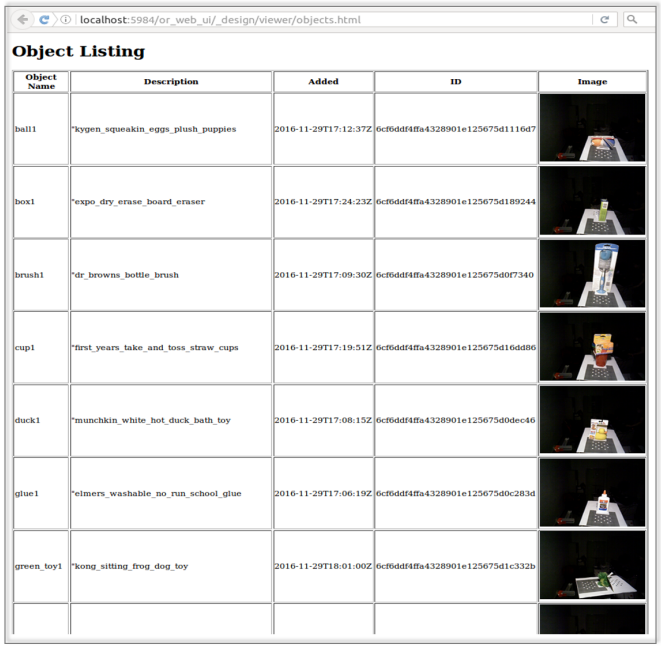


Figure 7.8: Screenshot capture of the creation of 3D meshes of objects. Left: Store objects 3D models in the database. Right: Objects' meshes contain (a) a glue, (b) a cup, (c) a box, (d) a spark plug, (e) a set of balls, (f) a brush, (g) a duck and (h)a pencil box.

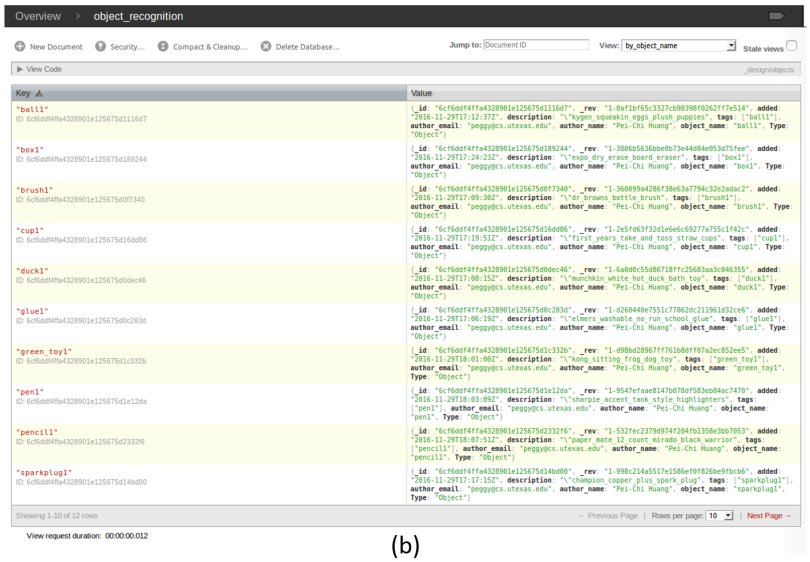
ject by parameters. Figure 7.9(a) illustrates a summary of meshes list for all objects. In Figure 7.9(b), left: each object name; right: this object id, added time, description, author name/email, and type. The 3D objects reconstruction in DB is used to predict 6D pose estimation for grasping.

7.2.3 Target 6D Objects Pose Estimation

The point clouds data was preprocessed to generate a model in 7.2.2 and this subsection introduces how to determine 6 DOFs pose estimation of



(a)



(b)

Figure 7.9: The object information in the CouchDB database. Screenshot captures the objects displayed as a document by the CouchDB administration tool. The search keys appear in the left box, and the values of each key appear to the right on the same line, including the object and the author information. The conclusion is that this database can be applied to predict 6D pose estimation for grasping.

a robot to grasp the target object. First, each model is generated the potential templates in 7.2.3.1. After the templates are built, given raw sensor data (depth and color features) as an input, and we require a method to match the filtered point clouds of the target objects with 3D models. A well-known approach, LineMOD [26, 27], exploits depth and color data to match the object's appearance and 3D shape in 7.2.3.2. LineMOD is one of the best methods for generic rigid fast template matching. The work combines with R-CNNs approach of predicting object location and pose estimation. Finally, 7.2.3.3 introduces iterative closest point (ICP[7]) and random sample consensus (RANSAC[8]) algorithms to refine the pose estimation.

7.2.3.1 Generate all Templates

A library *object_recognition_linemod* [88] was utilized to generate all templates for each object. This pipeline library implements LineMOD framework [26, 27] with an open source computer vision library – OpenCV library[9]. OpenCV [7] library is an open source library aiming at providing efficient computer vision algorithms and was applied to generate random views around an object. Given a mesh and different points of views based on angles, scale and in-place rotation of the camera as an input, and (1) depth (2) RGB (3) object size (4) distance between the object and the camera, from different angles perspective of 3D mesh object model, can be obtained and stored as templates

[7]ICP, `https://en.wikipedia.org/wiki/Iterative_closest_point/`
[8]RANSAC, `https://en.wikipedia.org/wiki/Point_set_registration/`
[9]OpenCV, `http://opencv.org/`

Figure 7.10: Snapshots are shown the generation of all templates for a glue. The input is a mesh; the outputs are depth, RGB, and the distance between the object and the camera from different scales and viewpoints. All the potential templates are stored in the database. The number of templates is below 100. Because the templates are used to match, if the cases are too many, it is difficult to judge. The conclusion is that to generate all templates can be used for templates matching.

in the DB. This process is repeated until enough coverage of the object yields from various viewpoints. The variety includes the different scales and points views with the in-plane rotations of the cameras. Figure 7.10 shows the process of generating the templates, and the object was rotated in-plane for each 20 degrees and also adjusted different scales to produce many samples. It is difficult to say how many templates are enough for matching. Based on our experiments, we found too many samples lead to failing. This is because too many similar samples make it difficult to distinguish which template is the best. Also, computation time is expensive. Therefore, for each object, we just generate below 100 templates.

7.2.3.2 Template Matching

Because a set of templates covering different views of an object was known, the 6D pose of an object can be coarsely estimated upon the DB. However, according to our experiments, their categorization results are not very precise due to a huge range of templates in the LineMOD pipeline. Therefore, to improve the accuracy of pose estimation and lower false positives, we applied the output from previous Subsection 7.2.1 which is a bounding box containing the target object. Since the type of object was detected and RGB-D information has been narrowed down, and the searching space of candidate templates has been lowered, so a template matching algorithm was implemented to acquire 6D pose estimation. The approach is to utilize surface normals and RGB gradient features to the object silhouette [91] and select from the DB. The output is the coordinates and orientation of one specific template.

7.2.3.3 Optimization of Pose Estimation

To enhance the accuracy of pose estimation, the resulting templates to point clouds set were been registered by using iterative closest point (ICP) algorithm [4, 50] and use random sample consensus (RANSAC) algorithm [88] to eliminate the mismatches points not belonging to the target area by finding the transformation matrix of features, and further find out the 6D pose of that object based on the geometric model. The target of point set registration is to get a spatial transformation that aligns two point sets. The ICP algorithm is one of the registration algorithms that tries to iteratively minimize the

difference between two point clouds. The ICP algorithm improves accurate results that can contribute to the success rate of the pick-and-place task. The snapshots, as shown in Figure 7.11, is an example of detecting a glue, and the outcome visualizes in a 3D visualization tool for ROS – RVIZ[10] to display object id, name and confidence, and each recognized object, visualizing point clouds and the mesh from the DB followed by a pose returned by LineMOD and refine pose estimation by ICP and RANDOM. However, the results are not good due to many noises from the environment, so we still combine with visual servoing alternatively.

7.2.4 Encapsulation to ROS package

Our aim is to create reusable software components for extending the capabilities of industry or academic robots. The perception module follows the guideline of the ROS framework [61] and is implemented as several ROS sub-modules. The ROS component-based platform provides more flexibility and modularity in facilitated development. All the aforementioned works such as object detection, recognition, 6D pose estimation are integrated as an automatic functionality on top of the ROS platform.

Next, we transfer object's coordinates and orientation for the manipulation module to guide the robotic arm performing a pick-and-place task.

[10]RVIZ, `http://wiki.ros.org/rviz`

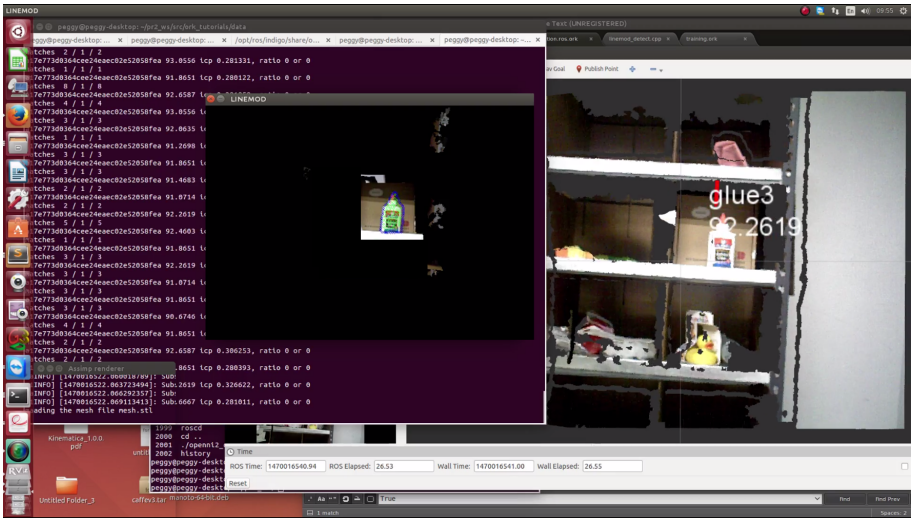


Figure 7.11: Snapshots of template matching. First, the glue was detected by using R-CNNs and circularized with a bounding box. Because the templates have been constructed, this object can be detected from the input point clouds by executing the LineMOD with ICP and RANSAC. The RVIZ plugin display window visualized the information of a glue, including object id, name, and confidence.

7.3 Manipulation Methodology

Recall that the 6D end-effector poses of specific objects from RGB-D information, and this section addresses the problem of planning a path between the initial pose of the robotic arm and the final desired pose for the gripper or suction. Figure 7.12 illustrates the procedure of manipulation module. Each action is decided by achieving a specific goal of manipulation. Therefore, the capability of the robotic arm can be reduced to a set of pre-defined motions from the known objects. The expected performance of the system should be both robust and fast while speeding up planning. After decided the gripper or suction, a collision-free path should be computed to guide the robotic arm from the current pose to pre-grasp postures. In the picking stage, the end-effector requires a sequence of joint trajectories that is produced for the robotic arm to follow based on visual serving or guided-policy searching method, and move the gripper to the post-grasp posture based on planning. In the placing stage, we move the gripper/suction from the post-grasp to the upper of the box and drop the grasped object.

7.3.1 Gripper and Suction

The target of the gripper is to end up grasping of the object in between the two fingers with a vertical parallel jaw. Because the information of the objects is known, a corresponding and advantageous grasp for each object such as the end-effector's forces can be pre-decided. The process of the gripper is described below: (1) Approach the object straight in the front rack facing

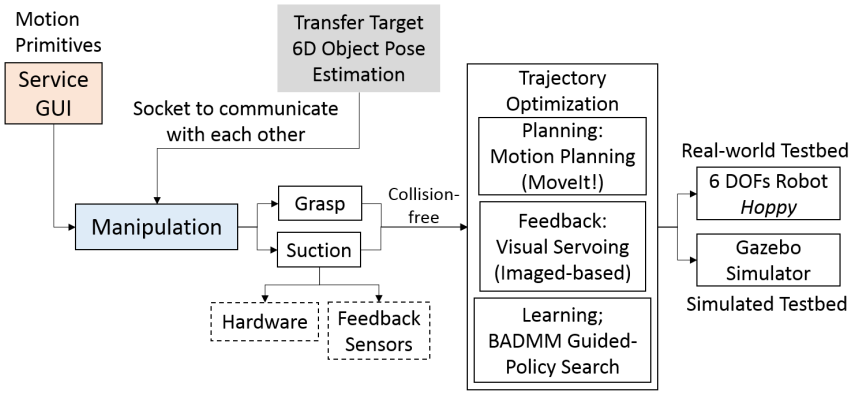


Figure 7.12: The flowchart of manipulation module in the pick-and-place system. The first procedure of this module is a gripper with two fingers and a suction design. Then, three approaches for trajectory optimization are motion planning (MoveIt!), visual servoing (Imaged-based) and learning algorithm (BADMM guided-policy searching). The process is implemented in the Gazebo simulation and then applied to a 6-DOFs robot *Hoppy* in real world.

the desired target (pre-action posture), (2) Find a collision-free path from pre-action to the final 6D pose estimations, (3) Open the grip to pick up the object, (4) Find a collision-free path from the final pose to the top of the box (post-action posture), (5) Open the grip to drop the object. However, the gripper still has many restrictions, for example, if the objects are placed in the bottom of the shelf, it is too close to grasp, so we still need a customized and universal gripper to make our system robust to exploit the diverse environment. On the other hand, the purpose of the suction is to deal with objects with a horizontal or vertical flat surface. The steps of the suction are as follows: (1) Move the robotic arm to the pre-actions posture, (2) Find a collision-free path from pre-action posture to the top centroid of the object, (3) Lower the suction to press the object, (4) Suck the object, (5) Lift up with the object if successful, (6)

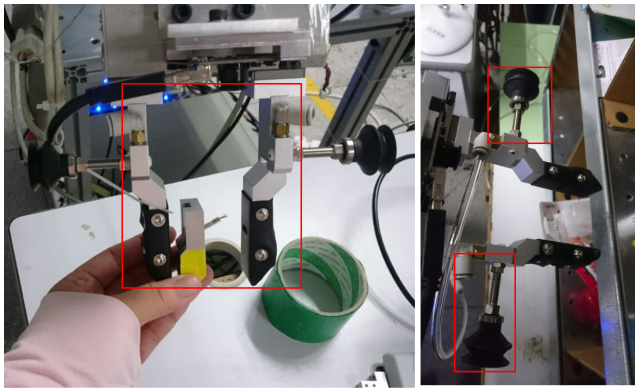


Figure 7.13: The customized design of the end-effector. (a) The long/short grippers with two symmetrical fingers; (b) The suction. Because of the variety of all APC objects, it is impossible to be associated with a particular scheme. Therefore, we design two types of end-effector.

Find a collision-free path from the $(5)^{th}$ position to the post-action posture, (7) Open the grip to drop the object. However, no force control of the gripper cannot sense when the arm should be stopped. Also, a plastic seal bags or clothes cannot work because their materials surfaces cannot be held by the suction. In general, a suction is easier than a gripper to pick up an object. Figure 7.13(a) shows the gripper with two fingers and (b) presents the suction. Some objects in APC are difficult to grasp, such as a book and a brush, so a suction through air pressure was designed to adhere to a nonporous surface, creating a partial vacuum.

7.3.2 Trajectory Optimization

This section first focuses on traditional planning methods for trajectories and adding vision control, likely visual servoing, and including learn-

ing approaches, such as Bregman alternating directions multiplier method (BADMM) guided-policy search. Such solutions will be discussed in more details. Those approaches aim to compute a collision-free trajectory or produce a general strategy/model to enable that the robotic end-effector can move between the two different conditions for picking and placing the target.

7.3.2.1 Motion Planning

Robotic motion planning is a maturing field of robotics. Once a valid 6D pose candidate is found, motion planning trajectories for 6 DOFs are generated and passed to the robotic controller to execute the movement. MoveIt! [83] is the widely-used software framework for motion planning in ROS and has been successfully integrated with many robots, such as the PR2 [90] and DARPA's Atlas robot. In the system, collision checking and motion planning were carried out with MoveIt! which uses plugins to enhance its functionality [12]: (1) Motion planning algorithm plugins: the Open Motion Planning Library (OMPL) [84], (2) Forward/inverse kinematics plugins: the OROCOS Kinematics and Dynamics Library (KDL) [79], (3) Detect collision: the Fast Collision Library (FCL) [57]. We used MoveIt! to constitute a collision-free path for the gripper or suction from one configuration to another while taking geometric or differential constraints into consideration. Unfortunately, although this approach can successfully compute collision-free paths, it is not done real-time. Because we have to wait for the production of a trajectory for each path and send back to the controller to execute, the approach is not fast and responsive

for the pick-and-place task.

7.3.2.2 BADMM Guided-policy search

Assume that an *agent* with Markovian system state x_t and action (control) u_t at time step t, $x_t, u_t \mid t \in [1, T]$. *Cost function* $l(x_t, u_t)$ defines the goal of task (i. e., distance between the gripper and the target). Thus, the total amount of cost is $\sum_{i=1}^{i=T} l_i(x_t, u_t)$. The current state x_t based on dynamics yields the next state x_{t+1}. A continually steps of (x_t, u_t) is a trajectory= $x_1, u_1, x_2, u_2, \ldots, x_T, u_T$. A *policy* defines the learning agent's way of behavior from perceived state to actions to be taken at a given time. Let $\pi(u_t \mid o_t)$ be a learned nonlinear global policy parametrized by weights θ, where o_t is observation at time step $t \in [1, T]$. Let $\pi(u_t \mid x_t)$ be a learned time-varying linear Gaussian mixture models (GMM) controller for initial state x_1^i.

In the pick-and-place task, our goal is to find a general trajectory policy $\pi(u_t \mid o_t)$ that an agent uses to choose actions u_t in a dynamical system. Each task is given by a cost function $l(x_t, u_t)$, and the objective is to minimize the expectation cost $E_{\pi_\theta}[\sum_{i=1}^{i=T} l_i(x_t, u_t)]$ over trajectory governed by the policy. However, traditional learning policies require numerous hand-engineered works for perception and low-level control (model-based techniques), so as to present that the policy relies on more representation of observed actions and states.

The searching method [45] combines a reinforcement learning algorithm and a supervised learning algorithm for trajectories policy decision. In general, supervised learning alone will not produce a good performance policy,

this is because the chained effects caused by the mistakes made on the policy part which places system states outside the unexpected distributions. To tackle the situation, Ross et al. [70] proposed that the training data should be generated from the state distribution of the original policy. By alternating between reinforcement learning and supervised learning, we can avoid the issue [45]. A reinforcement learning algorithm obtains trajectories distributions by generating samples from linear Gaussian controller on a physical robot. This stage can be formalized as Bregman alternating directions multiplier method (BADMM) algorithm [45, 87], a variant of ADMM [6], which is used to ensure the converged policy fits the same states of trajectories distributions [45]. A supervised learning algorithm for trajectories trains multiple policies $\pi_\theta(u_t \mid o_t) = N(\mu^\pi(o_t), \sum^\pi(o_t))$, where nonlinear function $\sum^\pi(o_t)$ defines a deep convolutional neural network (CNN) and nonlinear function (o_t) defines an observation learned covariance [45]. To learn complex tasks through the two techniques can be achieved by using cost function.

The policy search algorithm minimizes the expectation cost $E_{\pi_\theta}[l(\tau)]$, where τ is a trajectory and $l(\tau)$ is the cost for an episode. The main steps of the algorithm [45] are described below: (1). Acquire samples from linear Gaussian (GMM) controller $\pi(u_t \mid x_t)$; (2). Collect multiple images with initial/target positions, (3) Apply a nonlinear deep CNN policy $\pi_\theta(u_t \mid o_t)$ to match the local controller (the sampled distributions), as step (1) and (2) as input. (4). Estimate dynamics $p(x_{t+1} \mid x_t, u_t)$ for each linear Gaussian controller [44, 46], as shown in Figure 7.14. (5). Optimize local controller by

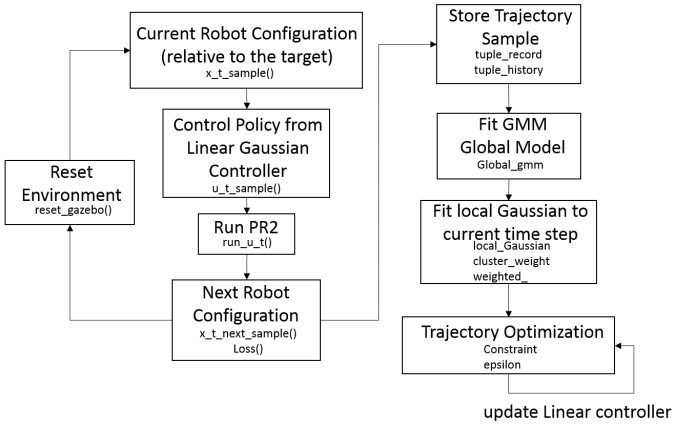


Figure 7.14: The flowchart of linear Gaussian controller on robotic arm.

modifying cost function to penalize deviation from the previous behaviors. (6). Update $\pi(u_t \mid x_t)$ with augmented cost. Repeat steps (1)-(6) until optimized policy parameters. Figure 7.15 shows a flowchart of the algorithm.

We implement the approach on the two robots. First, we use the Gazebo simulation with the model of Willow Garage Personal Robot 2 (PR2) embedded in its sensor system, which facilitates the simulation of robotic arm's behavior. We set 5 different conditions (paths) from initial to final positions, and for each condition the arm was repeatedly implemented 5 times, and each path was divided into 100 steps. We calculated average cost function and also pictured the results of trajectories, as shown in 7.16(a). The results prove that this model can be converged to one general model. Second, we construct *Hoppy* model in the Gazebo simulation, where simulates and exports the 6 DOFs' results to ROS, and we train different conditions to generate a general trajectory model, as shown in Figure 7.16(b).

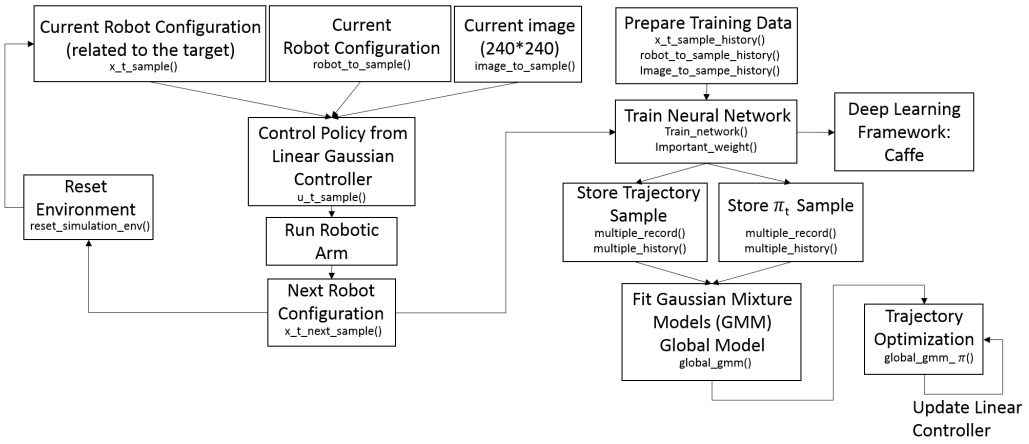
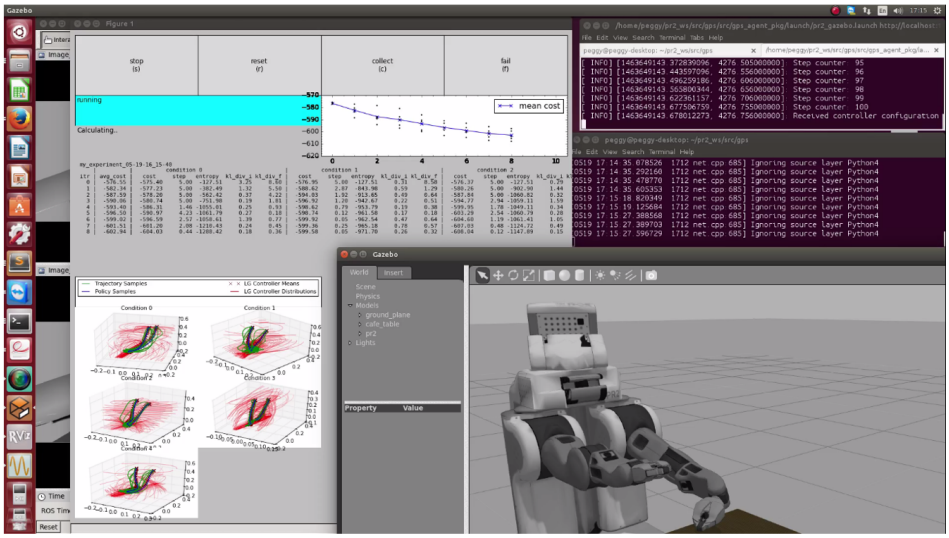


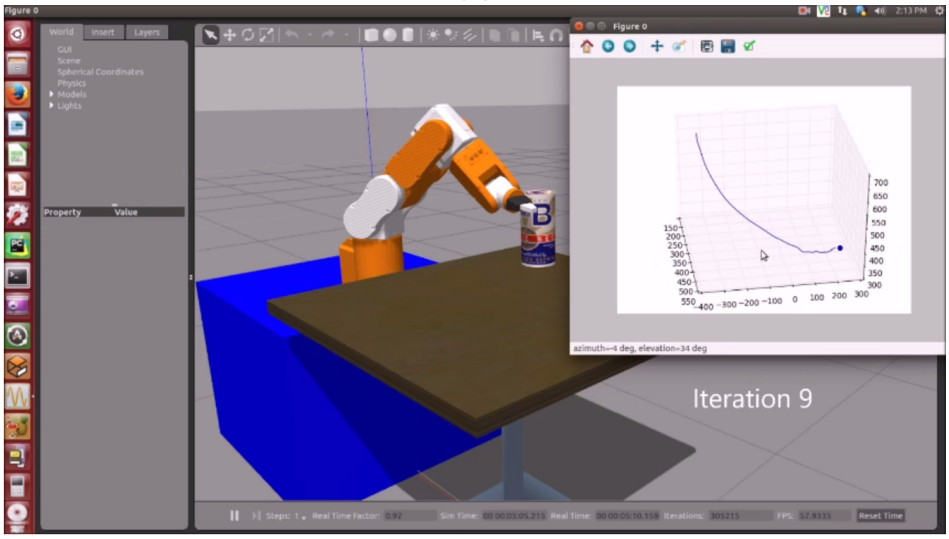
Figure 7.15: The flowchart of nonlinear Gaussian controller on robotic arm.

7.3.2.3 Visual Servoing

Recall that the techniques in 4.3, we implemented visual servoing for the pick-and-place task. A process to run visual servoing on robot is as follows: First, stereo cameras calibration: because OpenCV has no API to control focal lengths, a free software webcam application for the Linux desktop, GUVCview (GTK and UVC viewer), was adopted and modified to provide the functionality of controlling camera's focus, exposure and resolution in Ubuntu 14.04, and many photos were taken for calibration by exercising OpenCV samples code and we can get the mapping relations between the original images and rectified images. Second, hand-eye calibration: the same process as the previous step, and we can obtain the relationship between the robot and the camera. After obtaining intrinsic parameters (camera model) and extrinsic parameters (rotations and translations), the third step is to use Scale-Invariant Feature



(a)



(b)

Figure 7.16: Snapshots of the implementation of BADMM Guided-policy search in GAZEBO simulation. This approach was applied to different robots, PR2 and *Hoppy*. Despite using the different robots, the approach still can produce a general trajectory model.

Transform (SIFT[11]) algorithm to capture features on images to feed as our visual servoing program. Finally, the imported input data and features were sent to visual servo program to operate visual control for the robotic arm to the destination. We also implement visual servo in the Gazebo simulation through ROS and then transferred to *Hoppy* robot.

Figure 7.17 shows how we implement visual servoing. Figure 7.17(a) illustrates an architecture of visual servoing and how data flows through it, and Figure 7.17(b) illustrates how to convert OpenCV images to ROS format to be published over ROS. The Gazebo box is responsible for robot control and image fetching. *Hoppy* in the Gazebo simulation can use the cameras to capture RGB and depth streams, and calculate hand positions to publish as ROS topics. The visual servoing module box takes these topics as inputs by subscribing to them with ROS master service. After applying the aforementioned visual servoing, the component will generate the next robotic hand position/orientation and then send back to *Hoppy* to move the end-effector to the next position.

Figure 7.18 illustrates the implementation of visual servoing for different objects, including, a duck toy, a brush, a pair of glasses, a tennis ball, a set of pens, a box, a glue, and a water bottle. The bounding box in yellow represents the output from the visual servo approach colored in the red dot and the middle of the box colored in the green dot. The two points are eventually merged into one, and the direction of *Hoppy*'s arm toward the final grasping

[11]SIFT, `https://en.wikipedia.org/wiki/Scale-invariant_feature_transform`

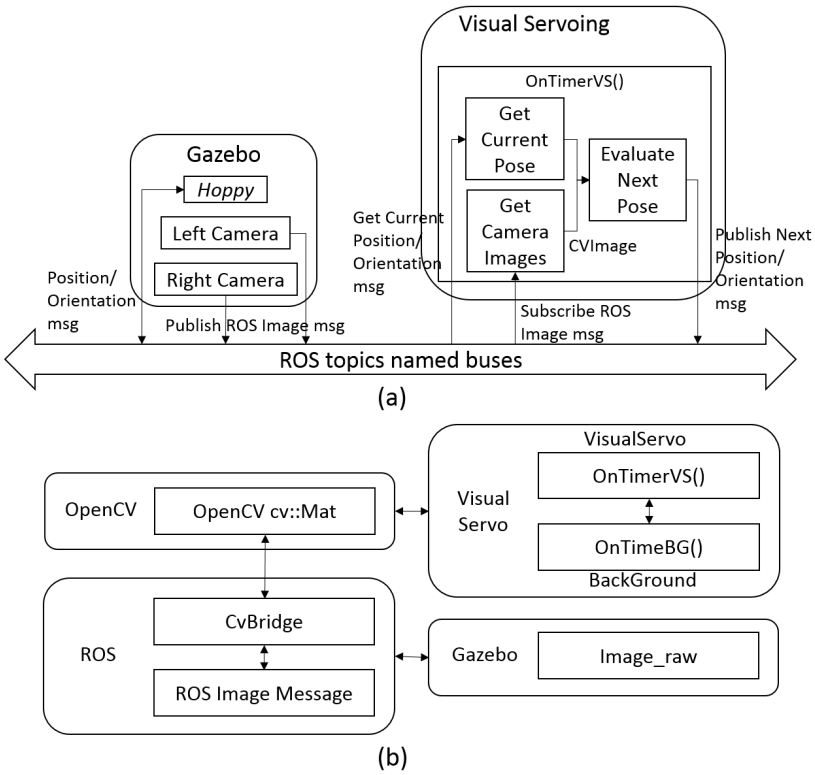


Figure 7.17: How visual servoing works. (a) An architecture of visual servoing and how data flow through it. (b). Convert OpenCV images to ROS format to be published over ROS.

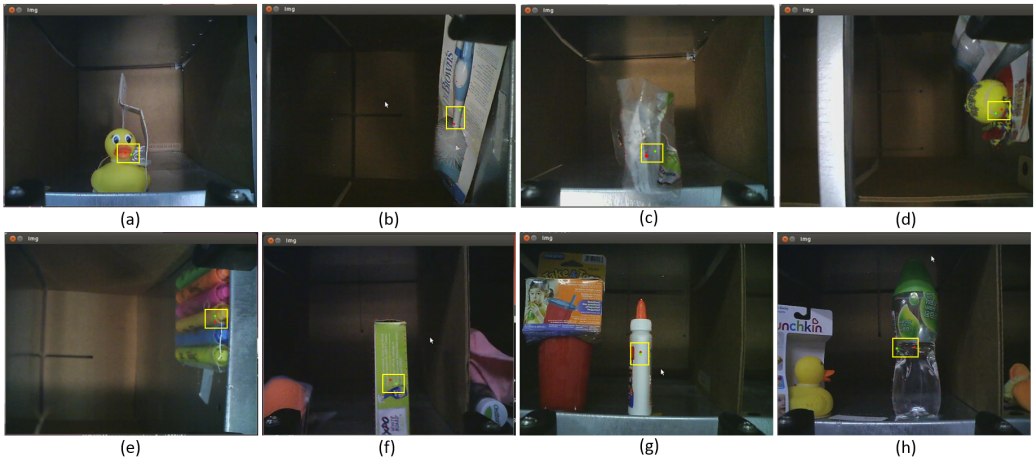


Figure 7.18: Implementation of visual servoing for 8 objects, (a) a duck toy, (b) a brush, (c) a pair of glasses, (d) a tennis ball, (e) a set of pens, (f) a box, (g) a glue, (h) a water bottle. The yellow bounding box indicates the red dot and the green dot; the red dot was returned by the result of visual servoing and the green dot points out the middle of objects. After iteratively executing the visual servoing approach, the two dots are overlapped, which means the robotic arm reaches the target. Then, the end-effector will perform the pick-and-place task for the desired object.

position.

7.4 Evaluation

This section evaluates our system on various scenarios and uses two grippers to grasp assorted items. We present our experimental setting in 7.4.1, and perform the pick-and-place task on the shelf with 18 objects in 7.4.2.

7.4.1 Experimental setting

The Amazon Picking Challenge competition (APC) in 2015 announced the competition for a general automatic pick-and-place system. We followed the rules to build an environment. Given a work cell 2×2 meters in front of a shelving unit populated with objects, and our task is to autonomously pick these items and place in the box. The 18 objects we used are obtained from the participants of APC, as depicted in Figure 7.4, including a duck toy, a brush, a tennis ball, a set of cups, a yellow toy, a green toy, a set of balls, wall plugs, pens, a box, cloth, a pencil box, two books, glasses, two bottles of water, and a sparking plug. The competition shelf is a standard Kiva pod used in the Amazon warehouse, and we also got it from the participants of APC in 2016. This shelf is composed of 12 individual bins. The difficulties are that each bin is not equal-size, so it is too deep to fetch, and also each bin has a lip which protects the objects from sliding, but the metallic bottom of the shelf produced the reflections. All factors have an impediment for the design of the pick-and-place system. Figure 7.19 illustrates that the environmental setting of the shelf, the 18 objects, *Hoppy* and a box will be placed on the shelf and the desk of *Hoppy*.

Recall that Section 7.2 mentioned, the perception module is the crucial component to identify the object and obtain 6D pose estimation with the equipped with the Xtion and the CMOS camera. We use the Xtion to get point cloud data (by measuring the time of light), including RGB-D ranging from 0.4 to 1.8 meters, and the resolution is 640×480. The experimental

Figure 7.19: The experimental setting for the Amazon Picking Challenge competition (APC): the shelf, 18 objects in Figure 7.4, and *Hoppy*.

hardware consists of (1) a manipulation controller PC (RTOS 12.02 Ubuntu) that controls *Hoppy* (2) a perception auxiliary computer (Ubuntu 14.04, GPU) that runs the operation for computer vision and learning algorithms. We use TCP/IP sockets to communicate with each other through WiFi. The perception software design is based on the middleware framework ROS, and work in the simulation Gazebo first, and RVIZ offers the visualization capability to observe the robot's behavior under a physics engine and environment. Finally, all are migrated to the real robot *Hoppy*. To get RGB and depth information, the Xtion camera is mounted on the end-effector of *Hoppy*. This camera is mounted on the pillar at the right-hand side, and RGB sensor at 30fps and the 640×480 depth resolution at 60fps. The depth sensor has $0.2 - 1.2$ meters range for the perception module.

First, a ROS cameras package of an open-source driver was developed.

Then, this package is applied to generate point clouds from raw depth map and RGB image utilizing existing UV map in real time. Next, to calibrate the mechanical relation between the camera and the robot, we developed a package for calibration. Then, to automate the pick-and-place experiment, luckily, we got some help to collaborate with the engineers from [69], and several commands were implemented on the control panel, as shown in Figure 7.20. The $X+$, $X-$, $Y+$, $Y-$, $Z+$, $Z-$ buttons can control the position of *Hoppy*'s end-effector; the $U+$, $U-$, $V+$, $V-$, $W+$, $W-$ buttons can manipulate the orientation of its. In PLC state input, the controller enables to start the suction cupule and then catch up the objects when pressed the button. After the Visual Servo button is pressed, *Hoppy* is directed to approach the object and the two finger motors are synchronized to perform the grasp.

7.4.2 Effectiveness of the Pick-and-Place system

The video[12] demonstrates the implementation of the perception and manipulation modules for the pick-and-place task. Besides, Figure 7.21 shows screen captures taken from a proof-of-concept demonstration of grasping 15 tasks. *Hoppy* can successfully approach target object before using vision and machine learning to recognize and obtain 3D location and 6D pose, and then grasp target objects when the controller through visual servoing, and place it in the box.

[12]The pick-and-place task, `http://www.cs.utexas.edu/~peggy/thesis.html`

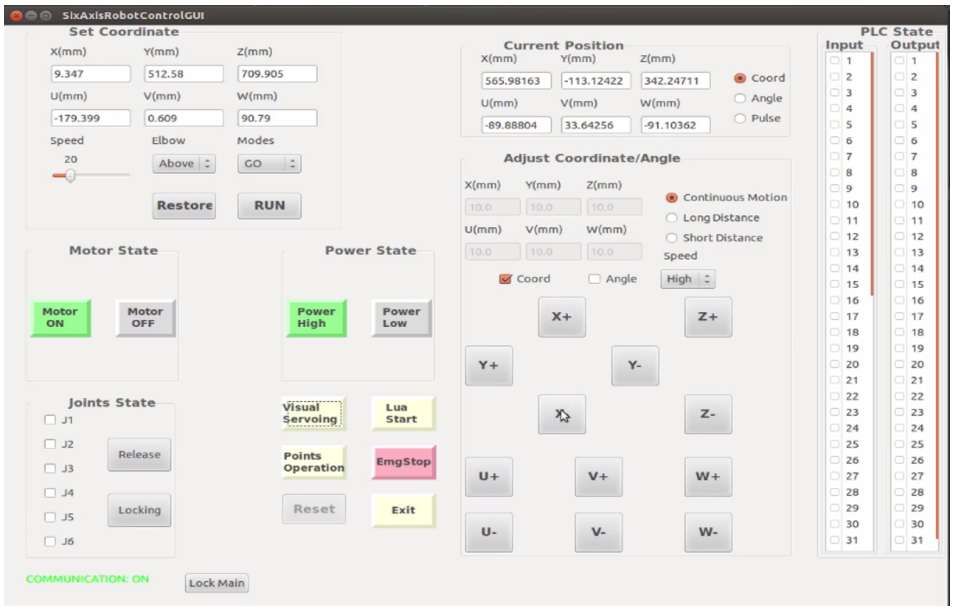


Figure 7.20: A screen capture of the panel of control software [69] for the pick-and-place task. In the default setting, *Hoppy* executes the gripper. However, if the PLC state input is marked, *Hoppy* works the suction. After the Visual Servo button is pressed, *Hoppy* is directed to approach the object and the two finger motors are synchronized or the suction cupule to perform the grasp. This panel can automate the pick-and-place experiment.

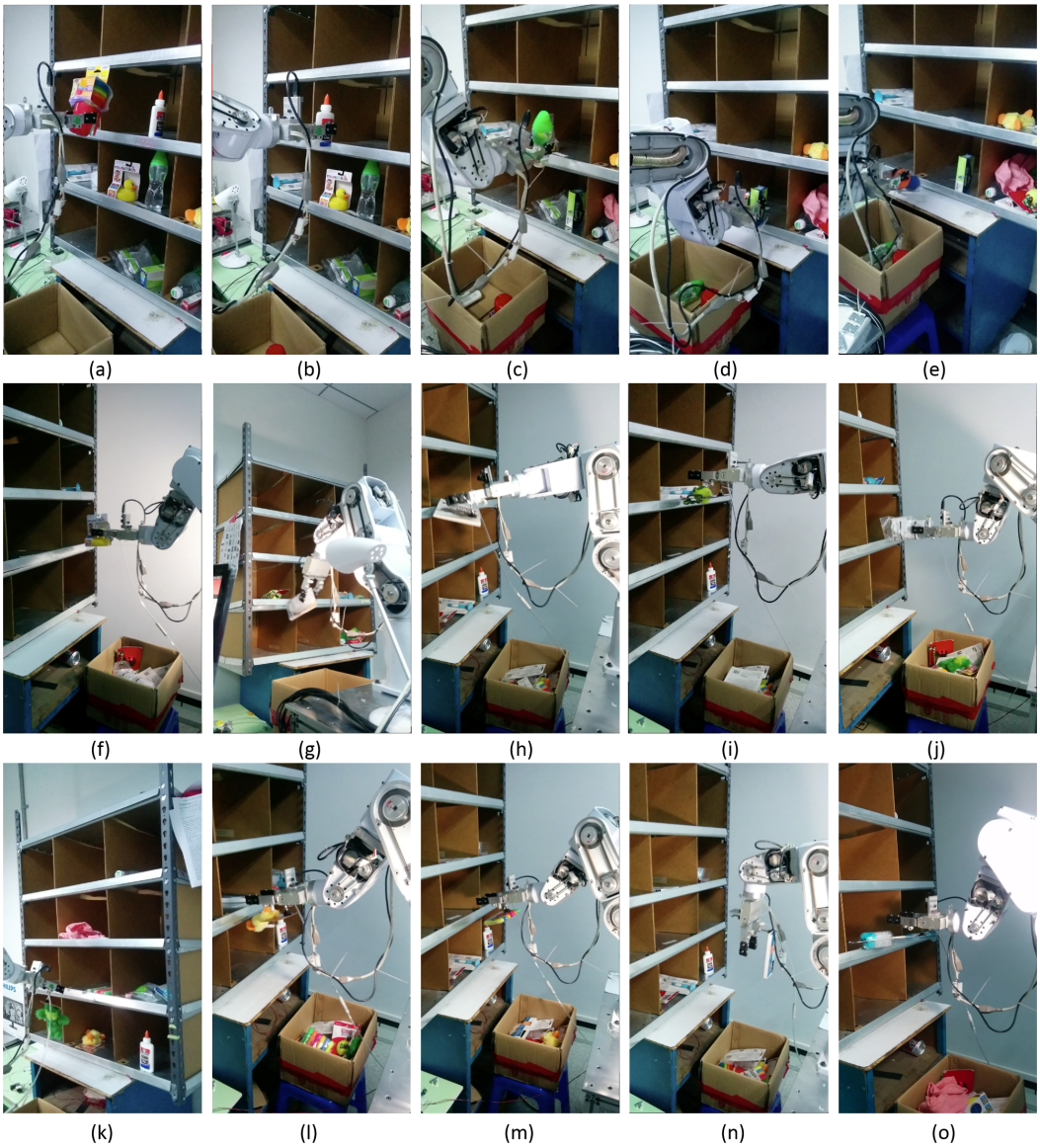


Figure 7.21: Our system picks and places several objects. The gripper grasped (a) a set of cups, (b) a glue, (c) a bottle of water, (d) a box, (e) a set of balls, (f) a duck toy, (g) wall plugs. The suction caught up (h) a book, (i) a tennis ball, (j) a pair of glasses, (k) a green toy, (l) a yellow toy, (m) pens, (n) pencils, (o) a brush. These figures confirm that our system can successfully accomplish the autonomous pick-and-place task.

The video link is `http://www.cs.utexas.edu/~peggy/thesis.html`

Chapter 8

The Real-time Grasping Performance Measurements

This section discusses whether string task completion deadlines can be met by using imprecise computation to trade increased speed for decreased accuracy. That is, in some situations a faster yet less precise grasp may better satisfy the use case. The grasping task can be divided into two stages. In the first stage, the robot moves its arm and body toward the target to be grasped. In the second stage, the robot assumes a posture that is best suited to grasp the target object and effects the grasp. The time the robot controller budgets to perform the grasping task directly bears on the outcome of the grasp. When considering tradeoffs, the design space of the grasping task has three main dimensions: (1) The training effort, measured by the time used in running the NEAT algorithm and deep learning to perform the grasp. (2) The task completion time, defined by the time Dreamer has to perform the grasping task. (To enforce the completion time constraint, a trajectory planner is used to compute the way-points for the trajectory that connects the initial and final configuration of Dreamer within the target completion time. The actual physical trajectory is realized by Dreamers and Hoppy's on-board controllers).

(3) The quality of the grasp is evaluated by simulation through the use of a fitness function and also by real experiments [31]. Because the grasp quality is the most important performance metric, the next sections explore tradeoffs between these dimensions. In 8.1, we first investigate how increasingly stringent time limits on computation reduce the accuracy of the robotic hand's approach trajectory. We then discuss the performance tradeoff between grasp quality and task completion time. In 8.2, we evaluate the tradeoff between training effort and grasp quality, and in 8.3, we measure the tradeoff between training effort and the task completion time with the successful grasp quality[1].

8.1 Grasp Quality vs. Task Completion Time Tradeoff Evaluation

For the grasping task, trajectories may be denoted by the point-to-point positions and orientations of the end-effector as long as no collision occurs. This section focuses on the actual interaction between the *Mekahand* and its environment assuming that there is no collision.

In a grasping experiment, the initial starting point S^* is the current position and orientation of *Dreamer's* end effector. The human supervisor assigns an object to be grasped from the user interface panel; the evolved neural network automatically determines the final destination D^* and orientation of *Mekahand* and sends it to *Dreamer's* main controller. On command, *Dreamer*

[1]This chapter is previously published in [31]. I contributed the experimental design, conduct and the performance evaluation to our work.

moves along the designated trajectory to approach and grasp the object, and then returns to the start position S^*. The actual trajectory of *Dreamer* is acquired by recording the position of the end-effector from forward kinematics calculations with joint positions. In controlling the movement of *Dreamer's* arm, we use the proportional-derivative (PD) controller in the Whole Body Control (WBC) algorithm. The position and orientation data are transmitted with a wireless system from the sensors to the control computer.

The first set of experiments measures the quality of the grasping trajectories versus various task completion times. In each experiment, *Mekahand* moves from S^* to D^* within a specified time interval of length ranging from 8 seconds down to 0.5 second. Each configuration was measured five times over different trials to obtain an accurate trajectory error estimation. An ideal trajectory was designed by a trajectory generation algorithm, and all experiments attempted to follow this trajectory, subject to different completion time deadlines. Each execution time was separately conducted five times, and averaged the five trajectories, as shown in Figure 8.1. In Figure 8.1, the highest variance in error are found in the 0.5 second trials, while the lowest variance in error are found in the 5 second trials. The differences between the ideal and actual trajectories for a one-way trip were recorded and depicted as boxplots for different time constraints and trials. Each scenario (execution time) was performed five times, and nine scenarios of experiments with the various execution times from 0.5 second to 8 seconds were tested. Therefore, forty-five results were yielded, as depicted in Figure 8.1 (a)-(i). Figure 8.1 illustrates that the five trials for

each execution time are comparable in the trajectory distributions. Figure 8.2 shows the difference between the ideal and actual trajectory over various task completion times. As expected, the trajectory closest to the ideal one is the one given the most time (i.e. 8 seconds). Figure 8.2 shows that in general, the shorter the completion time deadline, the higher the trajectory error. It should be noted that the design goal is to contain the trajectory error so that the grasp action can succeed at the end of the trajectory.

To predict the probability of success of a grasp, we can fit a statistical model to characterize the tradeoff between average trajectory error and task completion time. First, it is necessary to evaluate whether the five experiments are sufficient to represent ground truth. The standard deviation (STD) (root-mean-squared (RMS)) error for each task completion time respectively range from 0.00231(m) to 0.00009(m) (from 0.15742(m) to 0.01395(m)). Such a low STD indicates that the results do not vary much, and can therefore serve as a reasonable basis to derive models of RMS error.

In order to find a well-fitting regression model, Table 8.1 shows the results of approximated RMS error and adjusted R^2 of power, Weibull, rational, Gaussian and polynomial distributions. The trajectory errors are best modeled by a polynomial distribution of order five because of the lowest RMS errors and the highest R^2 values. The best fit polynomial model function $f_{\text{model}}(x)$

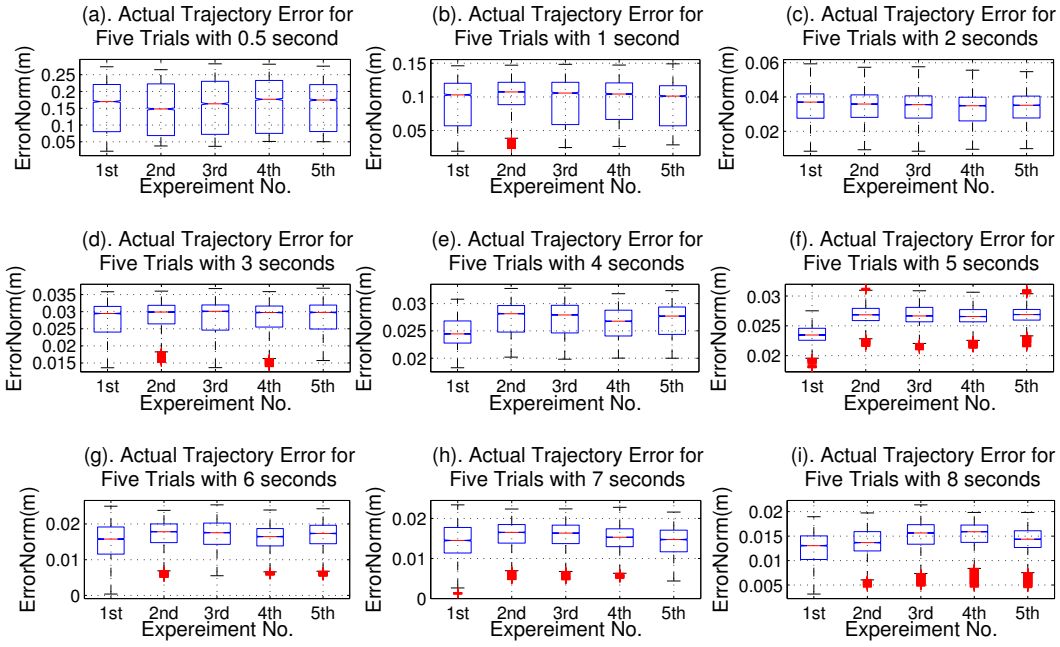


Figure 8.1: The results at five trials for nine scenarios with different execution times. The x axis indicates the trial number; the y axis indicates the normalized trajectory error compared to the ideal trajectory across the entire trajectory. Figures(a)-(i) show trials with execution times ranging from 0.5 to 8 seconds, summarizing in total the distribution of trajectory errors for 45 trials. The conclusion is that trajectory distributions for trials of particular length are similar enough to justify deriving statistics models.

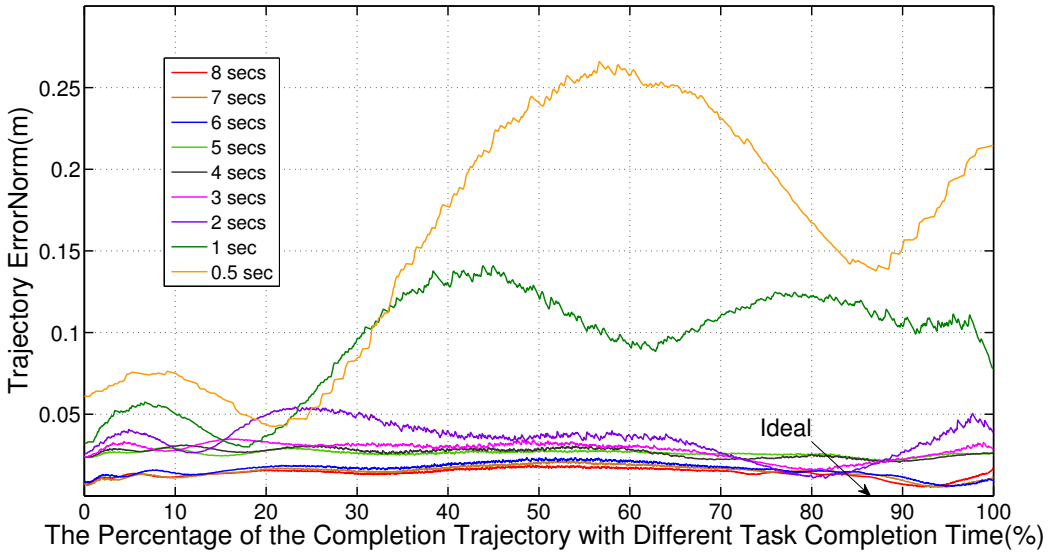


Figure 8.2: Tracking trajectories by varying execution times ranging from 8 to 0.5 seconds. The x axis represents the completion of the trajectory while the y axis represents the normalized trajectory error compared to the ideal trajectory. The trajectory error increases as the allowed time for execution decreases.

Fitting model	**RMS errors**	**adjusted R^2**
Power (2 terms)	0.006670	0.9812
Weibull	0.006179	0.9838
Rational (degree 5)	0.003164	0.9958
Gaussian (2 terms)	0.002755	0.9968
Polynomial (degree 5)	0.002308	0.9977

Table 8.1: A comparison is shown of five well-fitting statistical models of how RMS error varies with execution time budgets. The best fitting model is a polynomial of degree five because of the lowest RMS errors and the highest R^2 value, which is used to predict the expected RMS error at 9 seconds.

is:

$$\begin{aligned} f_{\text{model}}(x) &= -0.000714 * x^5 + 0.002011 * x^4 \\ &\quad + - 0.02155 * x^3 + 0.1088 * x^2 \\ &\quad + - 0.26 * x + 0.2635. \end{aligned} \tag{8.1}$$

Figure 8.3 shows the RMS errors fit to a linear interpolation and polynomial distributions. This model can predict the error after 9 seconds that is nearly stable (below 0.02 m) which indicates the 9 results are sufficient to proceed with the following experiment. To investigate how the reduction of task completion time may jeopardize sufficient accuracy for effecting a grasp, 6 seconds was chosen as the time-constraint for the following experiment.

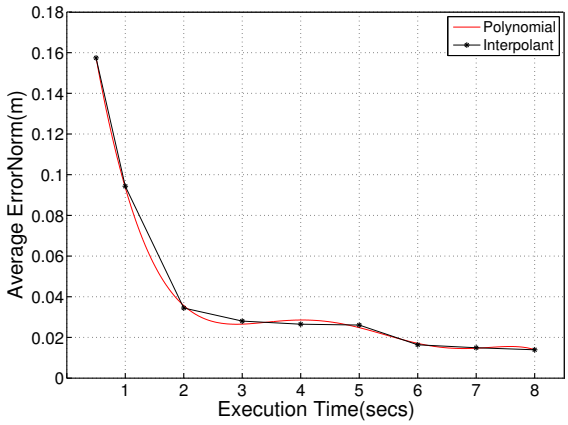
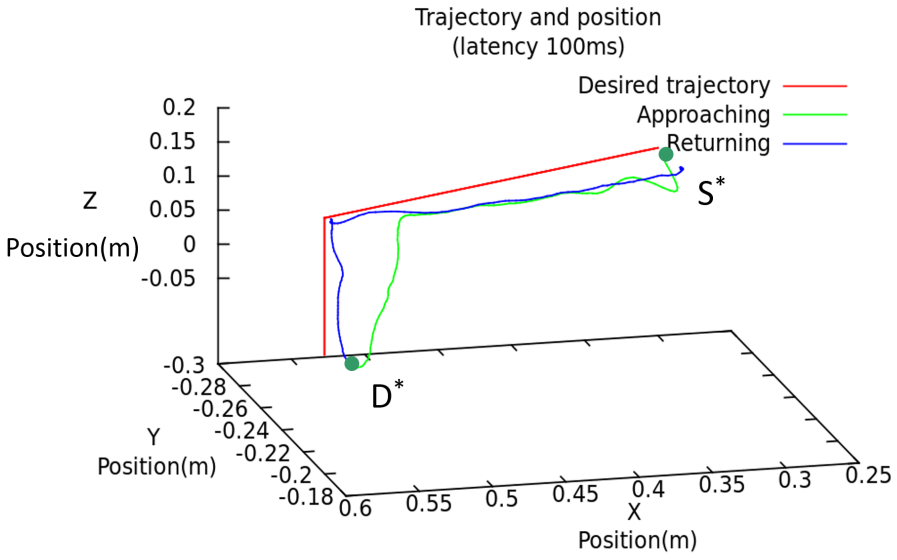


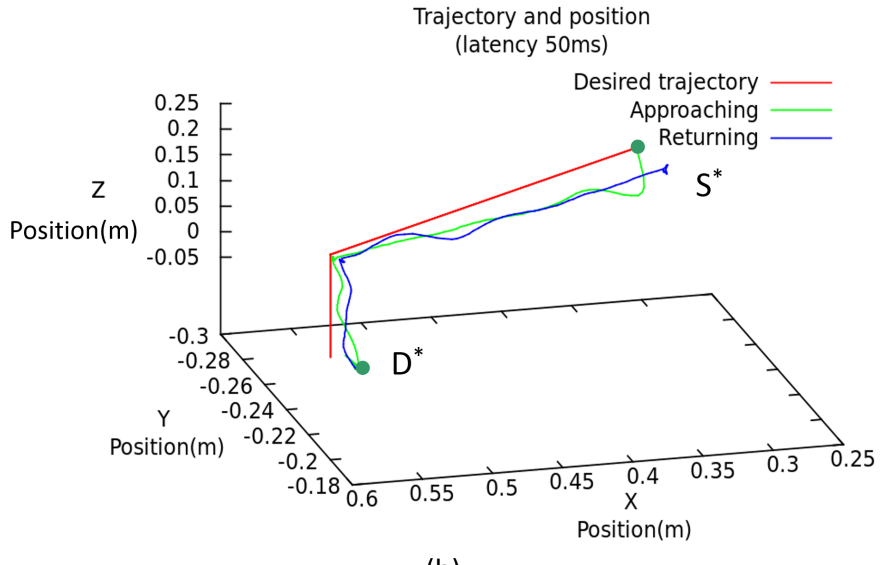
Figure 8.3: Fitting the RMS data with linear interpolant and 5-degree polynomials. This chart shows that inaccuracy is maximal when execution time is shortest (0.5 seconds), but rapidly improves as the budget increases to 3 seconds. Error decreases slightly between 3 and 6 seconds, and plateaus thereafter. The derivation line shows that the results approaches stability after 6 seconds, even the experiments after 9 seconds still can predict the error may be below 0.02 m.

The second set of experiments focuses on trajectory accuracy and latency delay within a given time constraint i.e. six seconds. The controller operation is the most time-consuming part of the practice, because many DOFs are considered in each step. In contrast, sending data and NEAT operation are relatively fast, only taking around 1-2 seconds each. Therefore, a round-trip is given fourteen seconds. The latency delays are 100ms, 50ms, and 10ms. Figure 8.4(a), (b) and (c) depict the relationships between the x, y, z position and the trajectory. The RMS errors with 100ms delay is 0.22487 meter(m); with 50ms is 0.21118 m, and with 10ms is 0.18203 m. The longer the latency delay, the worse the performance. In order to complete the task within the limited time frame, it is hard to control *Dreamer* very well. The accuracy is decreased when the latency delay is reduced. This is a design tradeoff between accuracy and latency delay. For further validation, Figures 8.5(a)-(i) depict the relationship between the position and time with 100ms, 50ms, and 10ms; Figures 8.5(j)-(l) display the relationship between error and time. Note that the error was computed as the actual minus the ideal desired trajectory. Figure 8.5, as expected, shows that the lowest latency delay (10ms) performs best.

The third set of experiments evaluates the success rate of grasp versus various task completion times. Based on the experiments, there is no influence between the grasp quality and task completion time. Even if we speedup the process, the grasp quality is almost the same, which means the controller is well-designed and stable. More experimental results are described in 8.3.


Trajectory and position
(latency 100ms)
Desired trajectory
Approaching
Returning
S*
D*
Z
Position(m)
0.2
0.15
0.1
0.05
0
-0.05
Y
Position(m)
-0.3
-0.28
-0.26
-0.24
-0.22
-0.2
-0.18
X
Position(m)
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25

(a)


Trajectory and position
(latency 50ms)
Desired trajectory
Approaching
Returning
S*
D*
Z
Position(m)
0.25
0.2
0.15
0.1
0.05
0
-0.05
Y
Position(m)
-0.3
-0.28
-0.26
-0.24
-0.22
-0.2
-0.18
X
Position(m)
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25

(b)

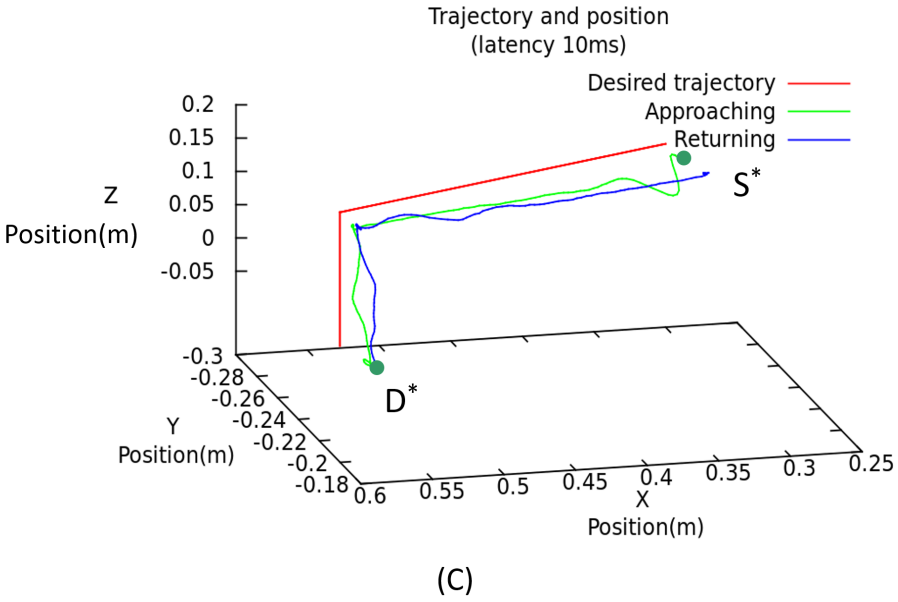
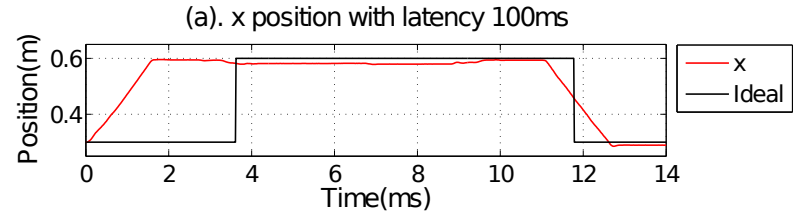
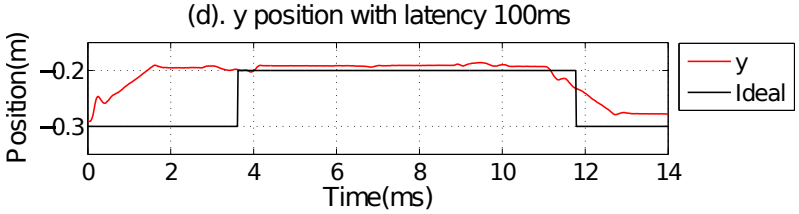
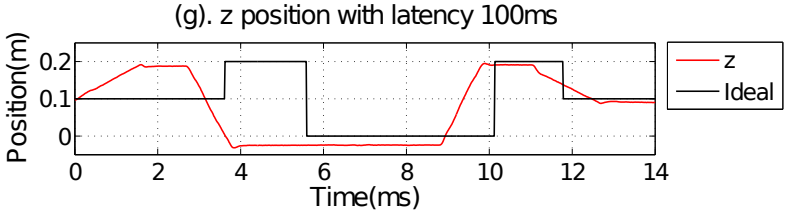
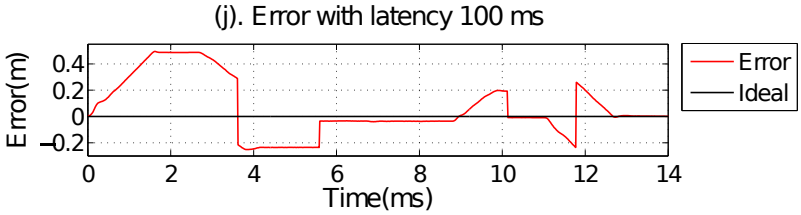


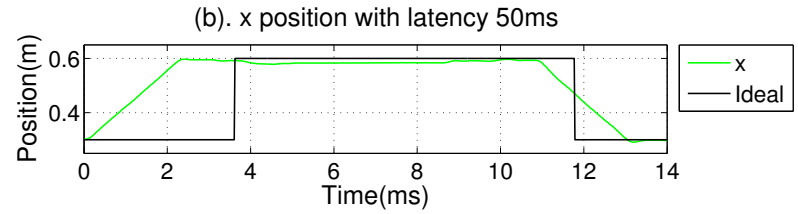
Figure 8.4: The trajectory for grasping using different latency delay of 100ms (a), 50ms (b), and 10ms (c). *Mekahand*'s end-effector starts from the point S^* to the destination D^* (the green line), and then returns to the origin state S^* (the blue line), and the red line represents the desired trajectory. The x, y, z represents the axes in 3D space where the end-effector moves. The errors with 100ms was the largest; the RMS errors with 10ms was the smallest. The conclusion is that the larger the delay, the worse the performance.

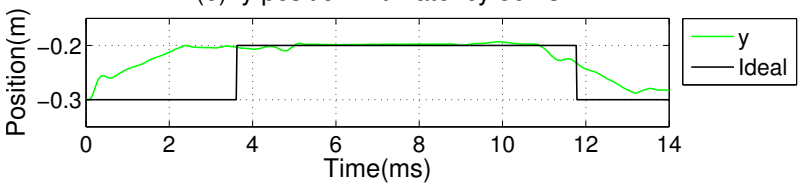

(a). x position with latency 100ms
Position(m)
0.6
0.4
0
2
4
6
8
10
12
14
Time(ms)
x
Ideal

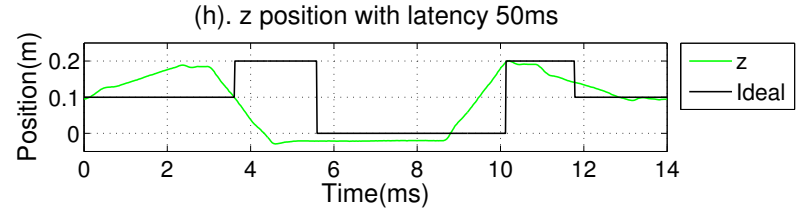

(d). y position with latency 100ms
Position(m)
−0.2
−0.3
0
2
4
6
8
10
12
14
Time(ms)
y
Ideal

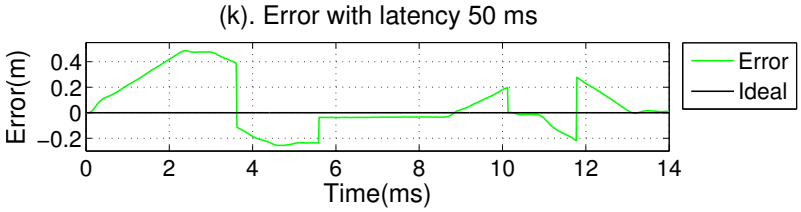

(g). z position with latency 100ms
Position(m)
0.2
0.1
0
0
2
4
6
8
10
12
14
Time(ms)
z
Ideal


(j). Error with latency 100 ms
Error(m)
0.4
0.2
0
−0.2
0
2
4
6
8
10
12
14
Time(ms)
Error
Ideal


(b). x position with latency 50ms
Position(m)
0.6
0.4
0
2
4
6
8
10
12
14
Time(ms)
x
Ideal


(e). y position with latency 50ms
Position(m)
−0.2
−0.3
0
2
4
6
8
10
12
14
Time(ms)
y
Ideal


(h). z position with latency 50ms
Position(m)
0.2
0.1
0
0
2
4
6
8
10
12
14
Time(ms)
z
Ideal


(k). Error with latency 50 ms
Error(m)
0.4
0.2
0
−0.2
0
2
4
6
8
10
12
14
Time(ms)
Error
Ideal

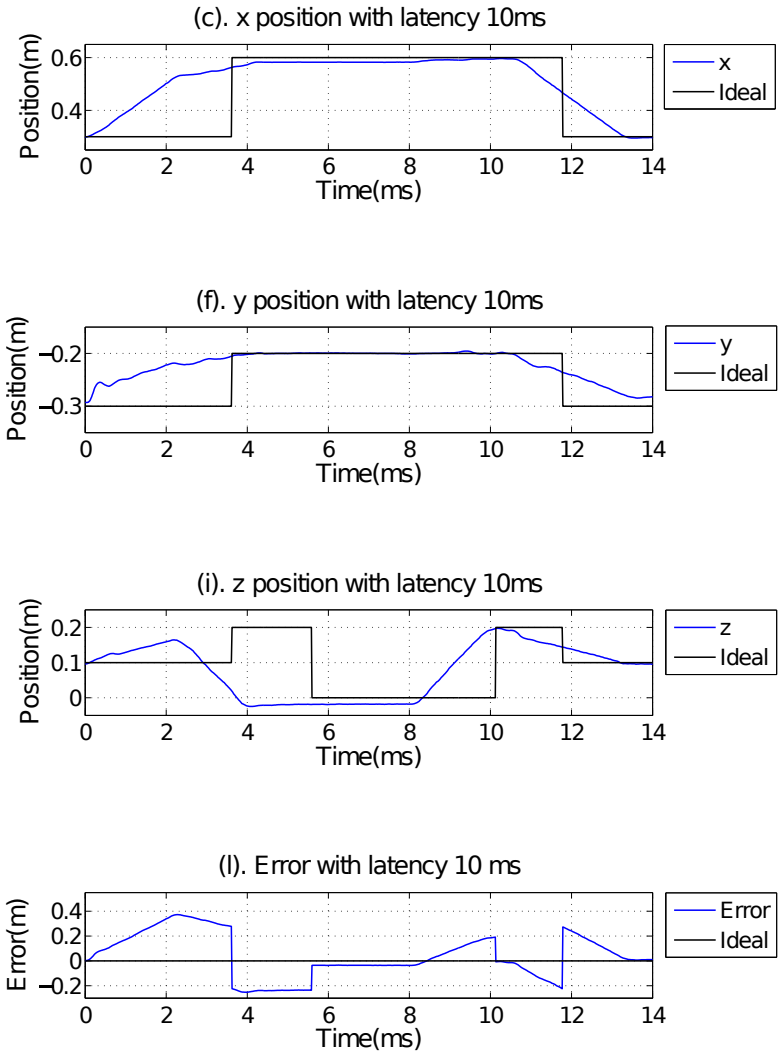


Figure 8.5: (a)(d)(g), (b)(e)(h), and (c)(f)(i) display the relationships between trajectory and latency delay (100ms, 50ms, and 10ms); (a)-(c), (d)-(f), and (g)-(i) show x, y, and z positions in 3D space. Figures(j)-(l) depict the relationships between the error and latency delay. The x axis represents the given time; the y axis in (a)-(i) denotes the actual and ideal positions, and in (j)-(l) denotes the error. Here, the error was calculated as the actual minus the ideal trajectory. As expected, the variation with 10ms was the best.

8.2 Training Effort vs. Grasp Quality Tradeoff Evaluation

We first describe the experimental setup and then present a set of grasping results that relate the quality of grasping to the training effort (defined to be the time spent on searching for the best *Mekahand* configuration for effecting the grasp by the NEAT algorithm). To speed up the training computation, we apply a parallelization strategy and run the NEAT algorithm with four multi-core computers.

We evaluate the effectiveness of our learning approach by conducting the two following sets of experiments. For the first experiment, the computational cost incurred by the *sequential* implementation is described in 4.2.2.2. The *parallel* strategy which dispatches different trials to all available computer cores is implemented to increase computational efficiency. In particular, work is dispatched over the network to multiple GraspIt! processes (thirty-six threads) which run on four computers, whose specifications are detailed in Table 8.2.

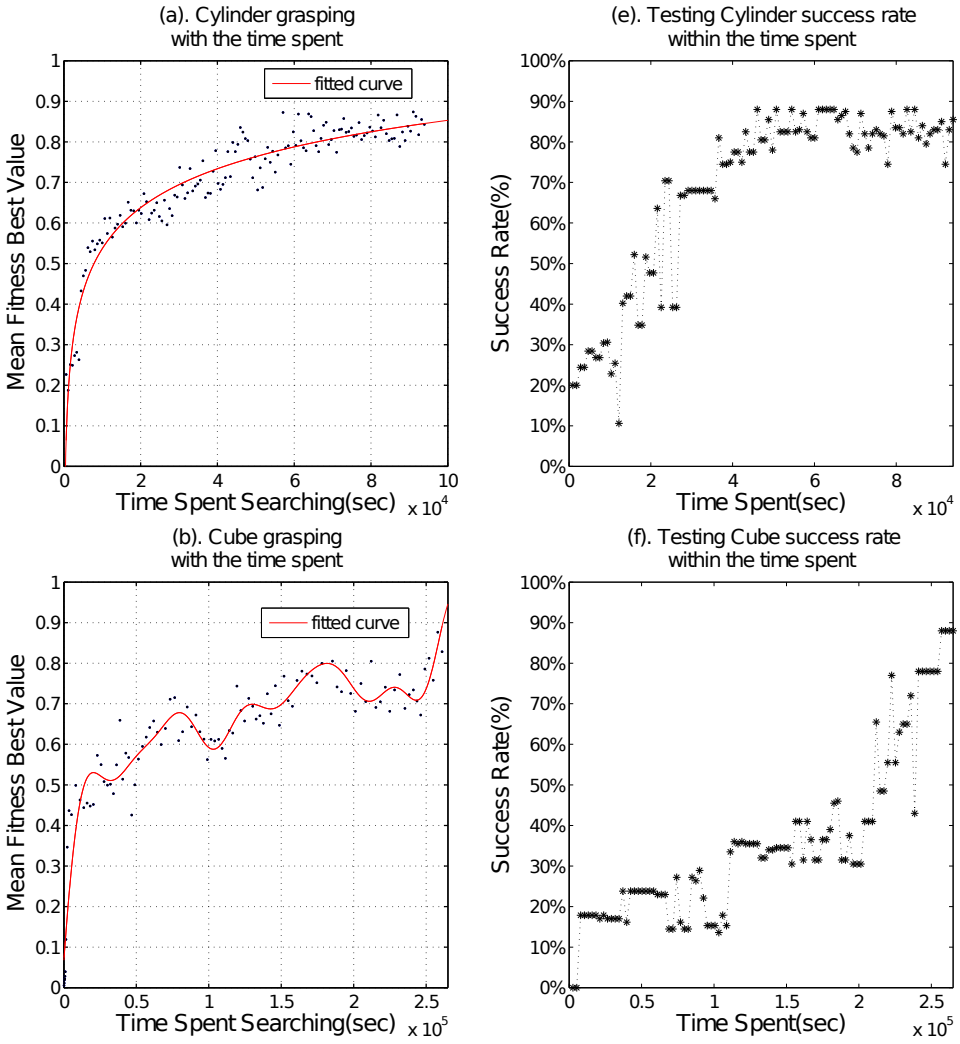
The first set of simulated experiments involve scenarios with different target objects, as explained in Section 6.1.3. Specifically, Figure 8.6(a)-(d) show the training results for networks trained to grasp a single cylinder, a single cube, a single sphere, and a single mug. Because the goal is to use only the best controller for the actual grasp, only overall best-case results are presented here. For each 10 minutes, the best-case value was recorded, and we repeated the same experiments three times, and the average values (the dots)

CPU	**GPU (NVIDIA GeForce)**	**Memory (DDR3)**
Intel Core i7 @ 3.6 GHz 4 Core / 8 Threads	GTX 760	16GB @ 1333 MHz
Intel Core i7@ 3.0 GHz 8 Core / 16 Threads	GTX 980	32GB @ 2133 MHz
Intel Xeon@ 2.67 GHz 4 Core / 8 Threads	GTX 285	8GB @ 1066 MHz
Intel Core i7@ 2.8 GHz 2 Core / 4 Threads	GT 520M	8GB @ 1333 MHz

Table 8.2: The experimental machine specifications. All of the CPU cores among the four computers can be fully employed by using the multi-threaded implementation, which significantly speeds up the evolution process.

and the fitting models are shown in Figure 8.6(a)-(d). These figures show how fitness values increase over evolutionary search. The underlying assumption is that larger fitness values implies better quality grasping.

To start the neuroevolution simulation, individuals in the population are initialized with random weights and a simple topology (i.e., in our case input nodes fully connected to four hidden nodes that are fully connected to the outputs), as shown in Figure 4.5. Because randomly generated policies generally do not cause the robot hand to approach the target objects, low fitness scores are expected. As time passes, Algorithm 1 leads the *Mekahand* to the right position/orientation toward the object. Accordingly, Figure 8.6(a), a cylinder, shows that initially the fitness value is low, and then after 80,000 secs, it reaches 0.8. Similar results appear in the other three experiments in Figure 8.6(b)-(d) (the cube, the sphere, and the mug). However, in Fig-


(a). Cylinder grasping
with the time spent
fitted curve
Mean Fitness Best Value
Time Spent Searching(sec)
x 10^4
(e). Testing Cylinder success rate
within the time spent
Success Rate(%)
Time Spent(sec)
x 10^4
(b). Cube grasping
with the time spent
fitted curve
Mean Fitness Best Value
Time Spent Searching(sec)
x 10^5
(f). Testing Cube success rate
within the time spent
Success Rate(%)
Time Spent(sec)
x 10^5

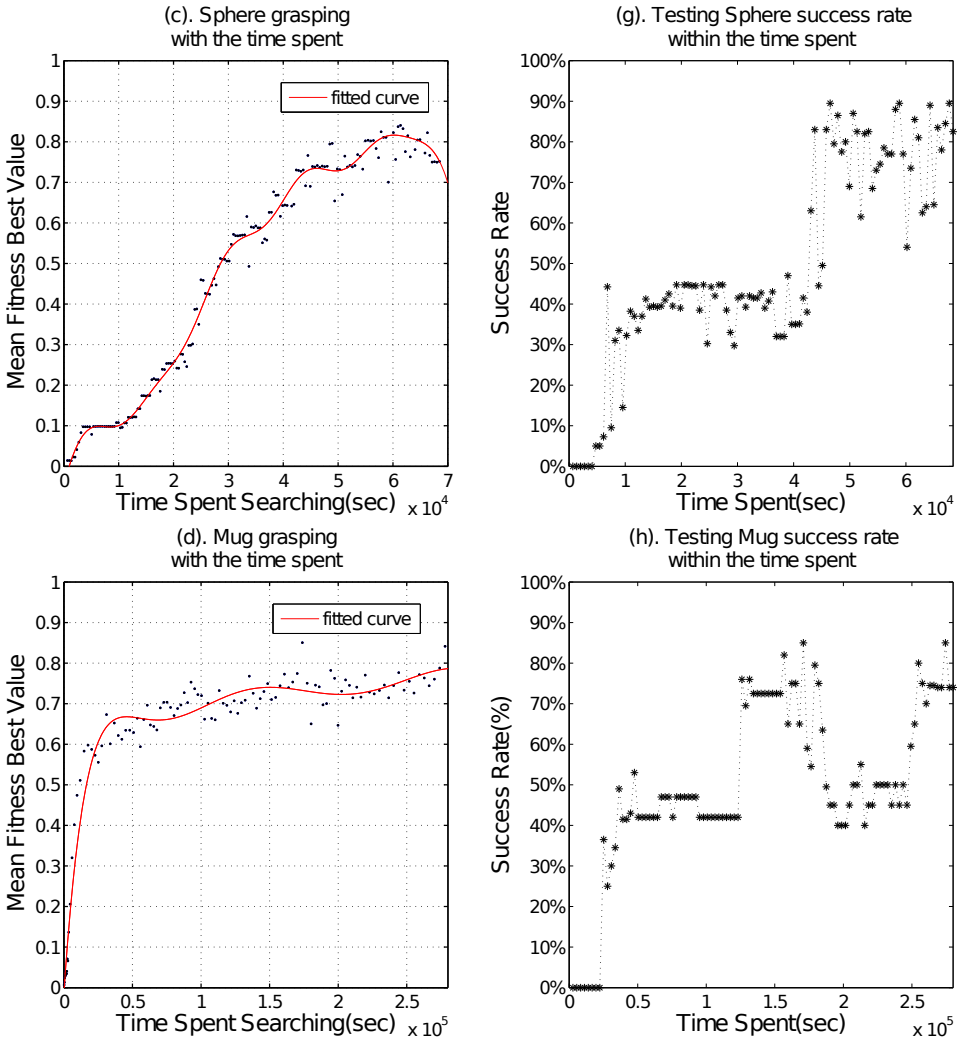

(c). Sphere grasping
with the time spent
fitted curve
Mean Fitness Best Value
Time Spent Searching(sec)
x 10^4
(g). Testing Sphere success rate
within the time spent
Success Rate
Time Spent(sec)
x 10^4
(d). Mug grasping
with the time spent
fitted curve
Mean Fitness Best Value
Time Spent Searching(sec)
x 10^5
(h). Testing Mug success rate
within the time spent
Success Rate(%)
Time Spent(sec)
x 10^5

Figure 8.6: Training and testing performance with the different time spent searching. How fitness values increase over time is shown for each experiment. Plots (a) and (e) show a scenario with a single cylinder on a table, (b) and (f) a single cube on a table, (c) and (g) a single sphere on a table, (d) and (h) a single mug on a table. To evaluate whether Algorithm 1 benefits performance, (a)-(d) are training results, while (e)-(h) are testing outcomes. Fitness generally improves as training progresses. The conclusion is that although accuracy generally benefits from increased time, increased time may sometimes also there is risk overfitting to the training cases, as shown by the intermediate trough in test performance in (h).

ure 8.6(b), (d) (the cube and the mug), the ANNs took 170,000 secs to get 0.8. In Figure 8.6(c), the sphere,the ANNs only used 50,000 secs to achieve 0.8. Two interesting results are noteworthy here. First, in the case of the sphere (Figure 8.6(c), the sphere), fitness dropped to 0.7 after 60,000 secs of training; one explanation is that sometimes overfitting occurs, which reduces accuracy. Second, in the case of a mug (Figure 8.6(d), the mug), the fitness value plateaus after 100,000 secs. Such plateaus in evolutionary search often reflect a local optimum; the mug is likely difficult to distinguish from the other objects. Comparing the four figures, it can be seen that the neural networks trained on the 'simple' objects (Figure 8.6(a), (c), the cylinder and the sphere) were much easier trained than networks trained on objects of more complicated geometry (Figure 8.6(b), (d), the cube and the mug). Figure 8.7 shows an example of an ANN evolved for 100,000 secs to grasp the cylinder.

The second set of experiments applies the NEAT algorithm on the same four target objects but in novel situations unseen in training, as discussed in Section 6.1.5. Here, the one-hundred best-trained neural networks that result

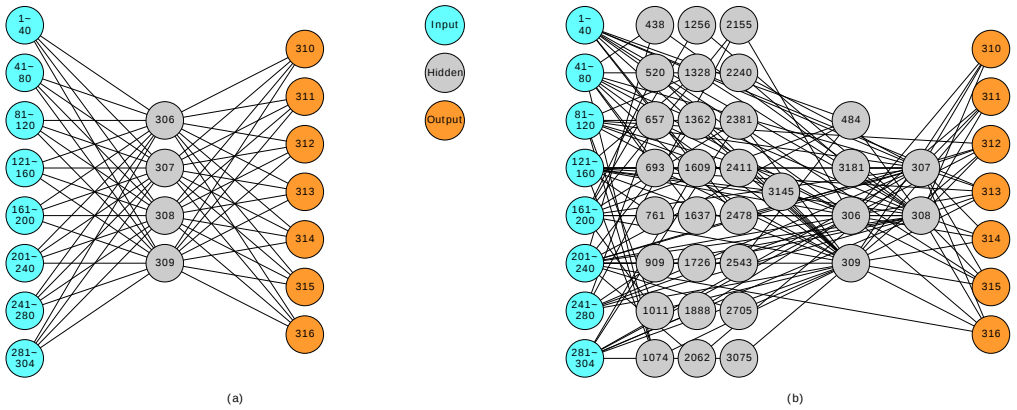


Figure 8.7: An example of a grasping network evolved by NEAT after $100,000$ secs. (a) The original ANN; the 304 Input nodes (20×15 pixels, a set of coordinate and object scale inputs) are located at the left-hand side, the 7 output nodes are located at the right-hand side. The light gray nodes in between are the evolved hidden nodes.(b) An example of a grasping network evolved by NEAT after 100,000 secs. The light gray nodes are evolved hidden nodes. The conclusion is that as time goes by, NEAT searches through increasingly complex networks to find one able to match the complexity of the grasping problem.

from different search times were picked and tested with the same input. Each ANN was tested 200 times. The success rates shown in Figures 8.6(e)-(h) compare generalization of neural networks versus time spent in training. The first experiment in Figure 8.6(e), the cylinder, shows that as the search time increases, the success rate improves and after 45,000 secs, the success rate reaches and stabilizes at around 80%. Similar results are seen in Figure 8.6(f), (g) (the cube and the sphere). In Figure 8.6(f), (g), the maximum success rate is 88%. However, in Figure 8.6(g), the sphere, the success rate continues to oscillate after 40,000, even after the training fitness value has stabilized around 0.8. This may reflect the difficulty in finding a robust and general grasp for an entirely-curved surface; maybe more training time is required. Figure 8.6(h), the mug, illustrates that sometimes grasps for complex objects can be consistent, although there is some oscillation in test performance later in evolution.

8.3 Grasp Quality vs. Training Effort vs. Task Completion Time Tradeoff Evaluation

This set of experiments measures the success rate of grasps across a two-dimensional surface of tradeoffs, examining how success varies across both training effort and task completion time. The idea is to explore what minimum amount of training effort and task completion time is necessary to successfully grasp an object. The grasping procedures were implemented under the designed motion planner, whose sequence consists of five states and state

transitions:(1) starting from initial to grasp coordinates, (2) grasp, (3) go to dropoff location, (4) place, and (5) go back to initial state, as illustrated in Figure 8.8. The different training stage of the evolved neural network were used to generate the grasping position and orientation of the *Mekahand* at step (1). A grasp was rated as successful if an object was grasped, lifted, and placed to another location from the initial position.

Figure 8.9 gives the complete grasping success rate with different task completion times and different training efforts. Each object was tested 10 times per task completion time and per training effort. For the implementation used, the grasping task completion time was measured ranging from 1 second to 10 seconds (the interval is 1 second) and the training effort was chosen ranging from 0 min to $1,200$ mins (the interval is 120 mins). Therefore, the total number of object tests is $10 \times 10 \times 10$. Overall, these results indicate that when accumulated training effort is higher than 600 mins, we could achieve 70% successful grasps rate. Interestingly, the task completion time has only a negligible impact on the success rate; even if only one second is allocated to task completion, the grasp quality remains nearly the same, indicating the controller has a stable design. The best match is depicted in Figure 8.9. Best performance results when the ANN is trained for 600, and the task completion time is 8 seconds.

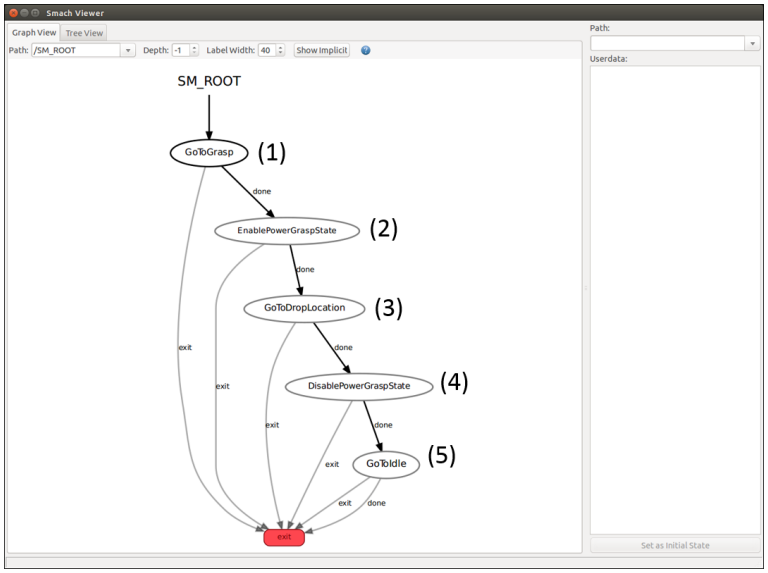


Figure 8.8: The designed trajectory includes five states: (1) starting from the initial state to the final grasping state based on the coordinates, (2) grasp, (3) go to dropoff location, (4) place, and (5) go back to the initial state. To following the trajectory, each run was tested if this grasp was successful.

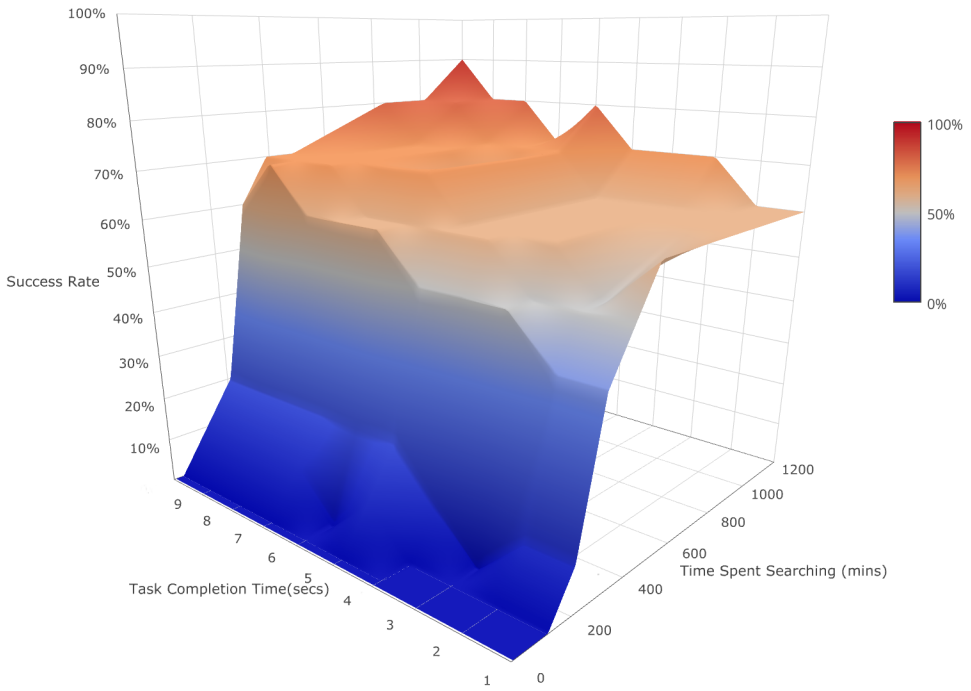


Figure 8.9: The grasp success rate vs. task completion time vs. training time. The x axis represents the task completion time, which ranges from 1 to 10 seconds, the y axis indicates the time spent searching ranging from 0 to $1,000$ mins, and the z axis represents the resulting grasp success rate. In general, the results indicate that a 70% successful grasp rate could be achieved after around 600 minutes of training effort. Moreover, it helps highlight what task completion time is sufficient to successfully grasp an object given enough training effort.

Chapter 9

Future Work and Conclusion

In the thesis, we present a cyberphysical avatar framework, defined as a semi-autonomous robotic system that exploits (1) body-compliant control in robotics, (2) skills acquired from machine learning, and (3) vision-based control of end-effector. Inasmuch as the research is about the integration of techniques, we expect to enable a robot can perform a particular type of task without human supervision. From a bottom-up perspective, teleoperation requires little intelligence but demands full participation from a human operator in real time. This is neither always possible (e.g., in a distributed real-time environment) nor desirable (e.g., operator fatigue). On the other hand, from a top-down perspective, full autonomy requires robot intelligence that is beyond the reach of the states of art. Our paradigm in this thesis is to adopt an approach that is in between and addresses a paradigm from fully manual toward autonomous robotics control for a selected type of tasks. Figure 9.1 shows a roadmap from hand-programmed to real-time automation. The ultimate goal is that fully autonomous robots can carry out mission-critical and safety-critical operations. Our research will not automate the determination of the type of human assistance required in a general setting. Rather, we want to provide an incremental

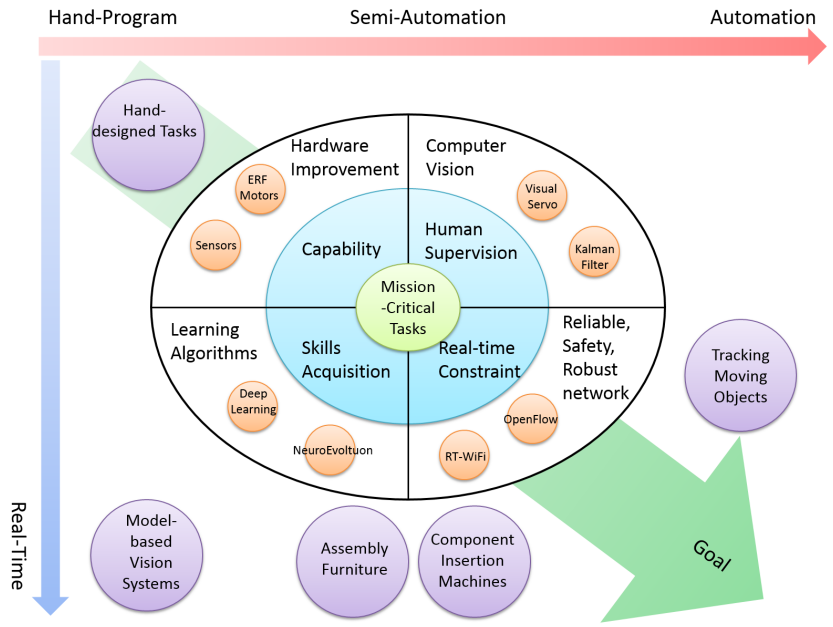


Figure 9.1: A roadmap toward generality for the robotics system. In the future research, the four components: capability, human supervision, skills acquisition and real-time constraints, can help robots to automatically accomplish many tasks under harsh situations.

path and the first step toward this ultimate goal. This chapter first presents the lessons learned and directions for future research in Section 9.1 and a brief summary of the thesis in Section 9.2.

9.1 Lessons Learned and Directions For Future Work

There are several interesting observations we may infer from this trade-off study. In the introduction, we asked whether the quality of grasping may exhibit a steep improvement after some critical number of training cases. As we can see from the results in Figure 8.6, the answer depends on the shape of

the object being grasped. From the results, the fitness function does not seem to be a very reliable predictor of the success of a grasp, especially when training time is limited. A possible explanation of the results is that for objects of shapes that generate a search space with sharp local maxima, the NEAT algorithm does not perform well until the neural network evolved assumes a topology that is compatible with the geometry of the search space, i.e., the neural network implements the correct function for some choice of the edge weights. From that point onward, the search should converge quickly to the correct weight configuration. If this interpretation has merit, then it would make sense to focus on searching for the right topology rather than assigning the right weights in the early phase of learning. A good heuristic for selecting the right network topology for the robotic task of interest would be especially useful, perhaps by employing ideas from simulated annealing. Further work is needed to better understand this issue.

A more practical result of this dissertation is the observation that the imprecise computation framework is a good match for robotic skill acquisition. Inasmuch as the goal is to grasp an object well enough so that the object cannot be dropped easily by minor perturbation on the controller, it is important to delineate the boundary between a good enough grasp and one that may drop the grasped object. The tradeoff results are useful for this delineation. For future work, it is useful to extend the tradeoff analysis to more complex robotic tasks than grasping.

More proposed studies in the dissertation are elaborated in next sec-

tions 9.1.1-9.1.2.

9.1.1 The Cloud-Based Computing for Real-Time Grasping Novel Objects

The rapid growth of cloud-based services contributes to the performance of high computation and the efficiency of data access distributed across the world. Here, we propose that the grasping task can be applied to cloud-based services. First, the data information from vision sensors can be uploaded to cloud-based services, and by means of cloud-based computing, speech recognition engines and object recognition/tracking engines can be generated. Then, grasp and path selectors also can be trained to produce end-effector configurations and trajectory engines. In the implementation of this task, followed by the preliminary object classification, these objects are tracked by the system with a bounded rectangle box on the control interface with tags. Then, the robot through speech recognition can be instructed to approach the selected object as voice input. Next, applying end-effector configurations and trajectory engine guides, the robot can perform a grasp. Figure 9.2 shows the integration of learning, control, and vision modules that are geared towards the task of picking up an object. Each technique is described below.

- **Vision Module:** Apply image pre-processing, object tracking, and recognition algorithms.
- **Learning Module:** (1) Given the desired objects' information taken from a vision module as an input for different learning algorithms, the

output is the final Cartesian position/orientation to which the robot's end-effector's moving plan gets an appropriate grasp. The grasper selection is continuously re-planned instead of completed with a one-shot implementation based on the robot's and object's current states in the control module. (2) Given the final Cartesian position and orientation of the end effector and the current state of the robot as an input for imitation learning, we will use forward simulations to decide on an appropriate Cartesian path (i.e., trajectory) that should be taken by the end effector from a current state to final state. The path decision could also be continuously re-planned in implementation based on an updated desired final state.

- **Control Module:** Apply visual servoing and ControlIt! [16] to control the robot. A feedback control component that fine-tunes the Cartesian position and orientation reference being supplied to ControlIt! based on vision sensing. By directly sensing the error in the current Cartesian position and orientation, visual servoing can (1) adapt to changes in the object's position / orientation and (2) account for modeling errors within ControlIt!. This will enable the robot's end effector to be positioned more precisely around the object and thus increase the probability of successfully picking up the object.

- **Real-Time Issue:** Real-time WiFi (e.g., RT-WiFi) may be used to communicate high-level task objectives like "grab that object" to the

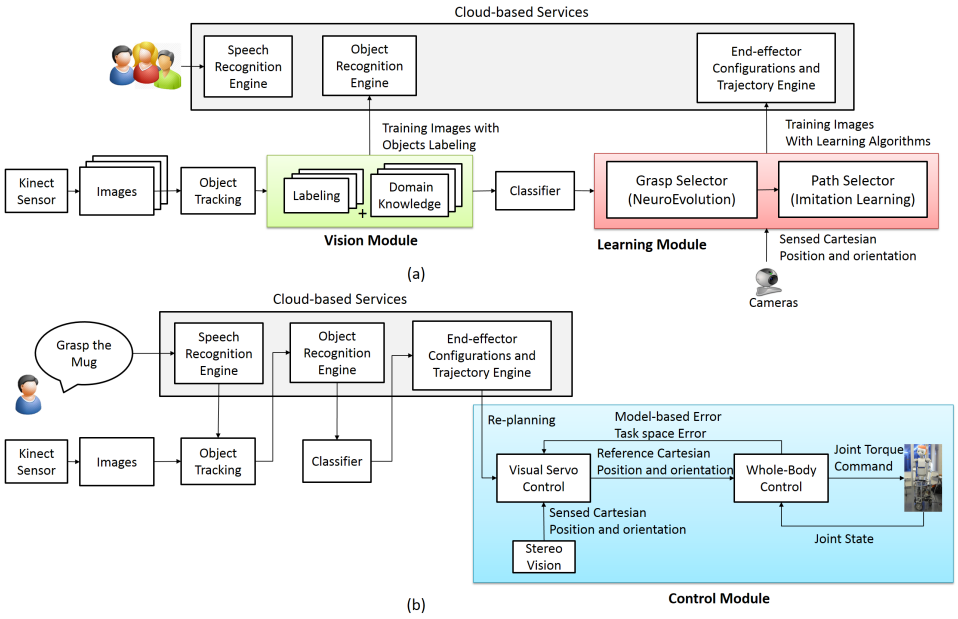


Figure 9.2: The cloud-based system framework for offline training phase in (a) and online testing phase in (b). Three modules are proposed: vision module for training objects tracking/recognition, learning module for training skills, and control modules for controlling robots in real world. Cloud-based services can speed up the training process.

robot, or as the communication medium between the components like the grasp selector and path selector that run at a slower frequency.

- **Cloud Services:** We also enable users to configure and reuse the formerly described system through a web browser interface or RESTful HTTP APIs and exploit the computational resources and coordination capabilities provided by the established infrastructure.

9.1.2 Apply HyperNEAT on Deep Architecture

Inspired by the recently reported success of the deep learning method, we propose to apply HyperNEAT to deep learning for the pick-and-place task. Like NEAT, deep learning's success seems to depend on the use of multi-layer neural networks with the proper edge connections for the target task. With the proper topology, a deep network may climb up the learning curve quickly with even a moderate number of training cases. So it is an interesting question to ask what the rate of task performance improvement is as a function of the training cases. A plausible conjecture is that with a properly connected multi-layer network, the performance curve may exhibit fast improvement once past some critical number of input training cases. For example, a juggling robot may "suddenly" acquire the juggling skill once some basic hand-eye coordination "invariant" has been captured by the evolving neural network.

HyperNEAT allows connection weights to vary across the phenotype in a regular pattern through an encoding called a *compositional pattern producing*

network (CPPN), which is an ANN but with specially-chosen activation functions. Although the HyperNEAT methods have been successful on a variety of tasks by utilizing geometric regularities [18, 80], so we could apply it to deep learning for robotic control which has shown promising results in the areas of object recognition. Therefore, we propose to use deep learning as an object feature learner while HyperNEAT as a grasping learner. Figure 9.3 illustrates our designed substrate logical connectivity for grasping. The logical connectivity for all the substrates compared experimentally. Neuron groups shown as connected are potentially fully connected. The input layer has four different types of image information, red, green, blue colors, and depth, and the total number of nodes is 40×30; the output layer has 7 nodes, including objects position and orientation. The input and output layers, except for topology, are the same as NEAT.

The multi-layer networks we proposed are illustrated in Figure 9.4. Our approach is described below. Given a set of visual input information, deep learning extracts features of objects from the input and classifies them. Then, HyperNEAT trains CPPNs to generate the connectivity for a designed ANN substrate. The ANN substrate processes the visual input information altogether with the features extracted by the deep learning to determine the best grasping posture of the objects.

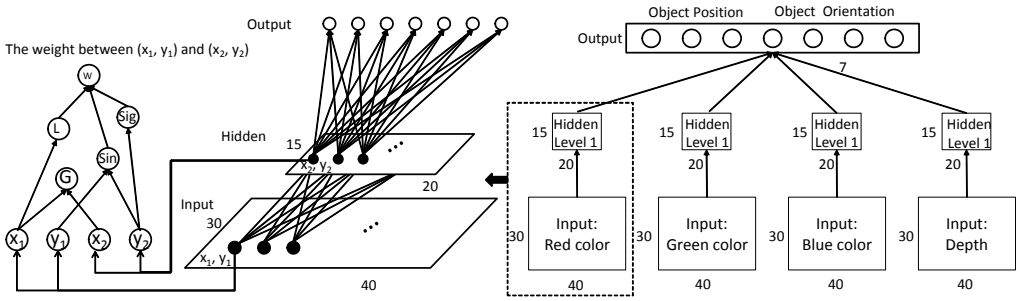


Figure 9.3: The designed substrate logical connectivity.

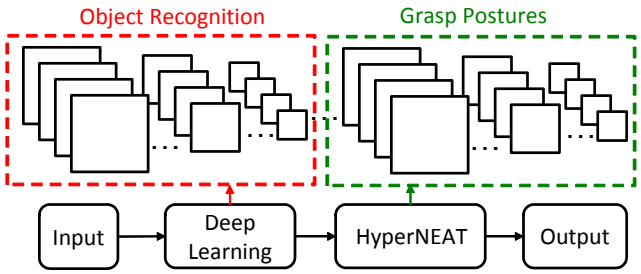


Figure 9.4: Deep Architecture Features Learning and HyperNEAT Grasping Learning. Deep learning recognizes the objects from the input and yield the extracted object features to HyperNEAT. HyperNEAT acts as a reinforcement learning approach that exploits in the geometric domain to determine the best way to grasp the objects.

9.2 Summary

Future robotic systems will need to function in unknown and unstructured environments, under demanding timing constraints. To meet the goal, this thesis presents a framework for realizing two semi-autonomous robots that integrate three key techniques: (1) whole body-compliant control, (2) skill acquisition from machine learning (neuroevolution methods and deep learning), and (3) vision-based control through visual servoing. We introduce a visual bounding box as an effective way for enhancing grasping performance, and also verify that transferring results from simulation to reality is possible even for the challenging problem of grasping unforeseen objects. This thesis proposes and demonstrates a systematically incremental approach to automating robotic tasks by decomposing a non-trivial task into stages, each of which may be automated by integrating the aforementioned techniques. We design and implement the controllers for two semi-autonomous robots that integrate three key techniques for grasping and pick-and-place tasks. Another contribution of the thesis is a tradeoff study in the design space over three key metrics: (1) the amount of training effort to teach the robot to perform the task, (2) the time available to complete the task once a command is given to perform it, and (3) the quality of the results of completing the task. In particular, we use the imprecise computation model, from the area of real-time systems research to systematically evaluate the performance of specific types of tasks: (1) grasping an unknown object and (2) placing the object in a target position in unstructured environments. The imprecise computation model helps us explore the

boundary region of error tolerance and evaluate best effort techniques. The tradeoff results in this study indicate that training effort is a critical factor for attaining high-quality grasping can function effectively in real time. The work here can be viewed as offering a realistic basis for the scheduling work done by the real-time systems community in the past two decades. The results were also validated with two real robots (*Dreamer* and *Hoppy*) as a step in the development of a systematic approach for designing robotic systems that can function in the challenging environments of the future, such as flexible manufacturing factories.

Appendices

Appendix A

Acronyms

[**APC**] Amazon Picking Challenge Competition

[**CAFFE**] Convolutional Architecture for Fast Feature Embedding

[**CNN**] Convolutional Neural Network, consist of multiple layers of receptive fields

[**CPS**] Cyber-Physical System

[**Gazebo**] Robot simulation for testing algorithms on designed robots in realistic scenarios

[**GraspIt!**] A simulator to accommodate arbitrary hand and robot

[**HyperNEAT**] The Hypercubed-based NeuroEvolution of Augmenting Topologies; extended from NEAT

[**LineMOD**] A fast template matching approach for generic rigid object recognition

[**NEAT**] NeuroEvolution of Augmenting Topologies; evolves artificial neural networks through an evolutionary algorithm

[**OpenCV**] Open Source Computer Vision Library is an open source computer vision and machine learning software library

[**PCL**] Point Cloud Library for point cloud processing and 3D geometry processing

[**ROS**] Robot Operating System provides libraries and tools to help software developers create robot applications

[**R-CNN**] Region-based with Convolutional Neural Network for deep features extraction

[**WBC**] Prioritized Whole-body Compliant controller

Bibliography

[1] The Cyber Pyhsical System Laboratory, Computer Science Department, The University of Texas at Austin. Available: `http://www.cs.utexas.edu/users/cps/`.

[2] Darknet: Open source neural networks in C. Available: `http://pjreddie.com/darknet/`. 2013–2016.

[3] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *International Journal of Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[4] Paul J. Besl and Neil D. McKay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

[5] Josh Bongard. The utility of evolving simulated robot morphology increases with task complexity for object manipulation. *Artificial life*, 16(3):201–223, 2010.

[6] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[7] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[8] Angelo Cangelosi, Gianluca Massera, and Stefano Nolfi. Evolution of prehension ability in an anthropomorphic neurorobotic arm. *Frontiers in Neurorobotics*, 1(4), 2007.

[9] François Chaumette and Seth Hutchinson. Visual servo control. I. Basic approaches. *IEEE Robotics and Automation Magazine*, 13(4):82–90, 2006.

[10] Jeff Clune, Kenneth O Stanley, Robert T Pennock, and Charles Ofria. On the performance of indirect encoding across the continuum of regularity. *IEEE Transactions on Evolutionary Computation*, 15(3):346–367, 2011.

[11] Jefferson Coelho, Justus Piater, and Roderic Grupen. Developing haptic and visual perceptual categories for reaching and grasping with a humanoid robot. *International Journal of Robotics and Autonomous Systems*, 37(2):195–218, 2001.

[12] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a MoveIt! case study. *arXiv preprint arXiv:1404.3785*, 2014.

[13] Lingfeng Deng, WJ Wilson, and F Janabi-Sharifi. Characteristics of robot visual servoing methods and target model estimation. In *Proceed-*

ings of the IEEE International Symposium on Intelligent Control (ISIC), pages 684–689, 2002.

[14] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[15] Carlo Ferrari and John Canny. Planning optimal grasps. In *Proceedings of the IEEE International Conference on Robotics and Automation(ICRA)*, pages 2290–2295, 1992.

[16] Chien-Liang Fok, Gwendolyn Johnson, John D Yamokoski, Aloysius Mok, and Luis Sentis. ControlIt!–a software framework for whole-body operational space control. *International Journal of Humanoid Robotics*, 13(01):1550040, 2016.

[17] The Apache Software Foundation. CouchDB: Couch DataBase. Available: `http://couchdb.apache.org/`.

[18] Jason Gauci and Kenneth O Stanley. A case study on the critical role of geometric regularity in machine learning. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 628–633, 2008.

[19] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[20] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[22] Xavi Gratal, Javier Romero, Jeannette Bohg, and Danica Kragic. Visual servoing on unknown objects. *Mechatronics*, 22(4):423–435, 2012.

[23] Song Han, Aloysius K Mok, Jianyong Meng, Yi-Hung Wei, Pei-Chi Huang, Quan Leng, Xiuming Zhu, Luis Sentis, Kwan Suk Kim, and Risto Miikkulainen. Architecture of a cyberphysical avatar. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, pages 189–198, 2013.

[24] Simon S Haykin. *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA:, 2009.

[25] Alexander Herzog, Peter Pastor, Mrinal Kalakrishnan, Ludovic Righetti, Jeannette Bohg, Tamim Asfour, and Stefan Schaal. Learning of grasp selection based on shape-templates. *Autonomous Robots*, 36(1-2):51–65, 2014.

[26] Stefan Hinterstoisser, Stefan Holzer, Cedric Cagniart, Slobodan Ilic, Kurt Konolige, Nassir Navab, and Vincent Lepetit. Multimodal templates for

real-time detection of texture-less objects in heavily cluttered scenes. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 858–865, 2011.

[27] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Gary Bradski, Kurt Konolige, and Nassir Navab. Model based training, detection and pose estimation of texture-less 3D objects in heavily cluttered scenes. In *Asian Conference on Computer Vision*, pages 548–562. Springer, 2012.

[28] Kaijen Hsiao, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Grasping POMDPs. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4685–4692, 2007.

[29] Kaijen Hsiao and Tomas Lozano-Perez. Imitation learning of whole-body grasps. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5657–5662, 2006.

[30] Pei-Chi Huang, Joel Lehman, Aloysius K Mok, Risto Miikkulainen, and Luis Sentis. Grasping novel objects with a dexterous robotic hand through neuroevolution. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Control and Automation (CICA)*, pages 1–8, 2014.

[31] Pei-Chi Huang, Luis Sentis, Joel Lehman, Chien-Liang Fok, Aloysius K Mok, and Risto Miikkulainen. Tradeoffs in real-time robotic task design

with neuroevolution learning for imprecise computation. In *Proceedings of the IEEE International Conference on Real-Time Systems Symposium (RTSS)*, pages 206–215, 2015.

[32] Seth Hutchinson, Gregory D Hager, and Peter I Corke. A tutorial on visual servo control. *IEEE Transactions on Robotics and Automation*, 12(5):651–670, 1996.

[33] The Hypercube-based NeuroEvolution of Augmenting Topologies. HyperNEAT software. Available: `http://eplex.cs.ucf.edu/hyperNEATpage/`.

[34] Nick Jakobi. *Minimal simulations for evolutionary robotics*. PhD thesis, University of Sussex, 1998.

[35] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, pages 675–678, 2014.

[36] Ishay Kamon, Tamar Flash, and Shimon Edelman. Learning to grasp using visual information. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2470–2476, 1996.

[37] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[38] Nate Kohl, Kenneth Stanley, Risto Miikkulainen, Michael Samples, and Rini Sherony. Evolving a real-world vehicle warning system. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1681–1688. ACM, 2006.

[39] Danica Kragic and Henrik I Christensen. A framework for visual servoing. In *International Conference on Computer Vision Systems*, pages 345–354. Springer, 2003.

[40] Steven M LaValle. *Planning algorithms*. Cambridge University Press, 2006.

[41] Quoc V Le, David Kamm, Arda F Kara, and Andrew Y Ng. Learning to grasp objects with multiple contact points. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 5062–5069, 2010.

[42] Joel Lehman and Risto Miikkulainen. Neuroevolution. *Scholarpedia*, 8(6):30977, 2013.

[43] Joel Lehman, Sebastian Risi, David DAmbrosio, and Kenneth O Stanley. Encouraging reactivity to create robust machines. *Adaptive Behavior*, 21(6):484–500, 2013.

[44] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.

[45] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[46] Sergey Levine, Nolan Wagener, and Pieter Abbeel. Learning contact-rich manipulation skills with guided policy search. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 156–163, 2015.

[47] Hod Lipson, Josh C Bongard, Viktor Zykov, and Evan Malone. Evolutionary robotics for legged machines: From simulation to physical reality. In *Proceedings of the 9th International Conference on Intelligent Autonomous Systems (IAS)*, pages 11–18, 2006.

[48] Jane W.-S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang-shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, pages 203–249. Springer, 1991.

[49] Jane W.-S Liu, Wei-Kuan Shih, Kwei-Jay Lin, Riccardo Bettati, and Jen-Yao Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, 1994.

[50] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. *Chapel Hill, University of North Carolina*, 4, 2004.

[51] Andrew Miller, P Allen, V Santos, and F Valero-Cuevas. From robotic hands to human hands: a visualization and simulation engine for grasping research. *Industrial Robot: An International Journal*, 32(1):55–63, 2005.

[52] Andrew T Miller and Peter K Allen. Examples of 3D grasp quality computations. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 1240–1246, 1999.

[53] Andrew T Miller and Peter K Allen. GraspIt! a versatile simulator for robotic grasping. *IEEE Robotics and Automation Magazine*, 11(4):110–122, 2004.

[54] Andrew T Miller, Steffen Knoop, Henrik I Christensen, and Peter K Allen. Automatic grasp planning using shape primitives. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 1824–1829, 2003.

[55] Javier Molina-Vilaplana, J Feliu-Batlle, and J Lopez-Coronado. A modular neural network architecture for step-wise learning of grasping tasks. *Neural Networks*, 20(5):631–645, 2007.

[56] Richard M Murray, Zexiang Li, S Shankar Sastry, and S Shankara Sastry. *A mathematical introduction to robotic manipulation*. CRC press, 1994.

[57] Jia Pan, Sachin Chitta, and Dinesh Manocha. FCL: A general purpose library for collision and proximity queries. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3859–3866, 2012.

[58] Leonardo M Pedro, Valdinei L Belini, and Glauco AP Caurin. Learning how to grasp based on neural network retraining. *Advanced Robotics*, 27(10):785–797, 2013.

[59] Raphael Pelossof, Andrew Miller, Peter Allen, and Tony Jebara. An SVM learning approach to robotic grasping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 4, pages 3512–3518, 2004.

[60] Justus H Piater. Learning visual features to predict hand orientations. *Computer Science Department Faculty Publication Series*, page 148, 2002.

[61] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on Open Source Software*, volume 3, page 5. Kobe, Japan, 2009.

[62] Deepak Rao, Quoc V Le, Thanathorn Phoka, Morgan Quigley, Attawith Sudsang, and Andrew Y Ng. Grasping novel objects with depth segmentation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2578–2585, 2010.

[63] Simón J Rascovsky, Jorge A Delgado, Alexander Sanz, Víctor D Calvo, and Gabriel Castrillón. Informatics in radiology: use of CouchDB for document-based storage of DICOM objects. *Radiographics*, 32(3):913–927, 2012.

[64] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[65] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015.

[66] Shaoqing Ren, Kaiming He, Ross Girshick, Xiangyu Zhang, and Jian Sun. Object detection networks on convolutional feature maps. *arXiv preprint arXiv:1504.06066*, 2015.

[67] Nasser Rezzoug and Philippe Gorce. *Robotic grasping: A generic neural network architecture*. INTECH Open Access Publisher, 2006.

[68] Robotics Lab, Computer Science Department, Columbia University. GraspIt! Software. Available: `http://www.cs.columbia.edu/~cmatei/graspit/`.

[69] Robotics Robotics Ltd. The robotic arm *hoppy*. `http://www.robotics-robotics.com/`.

[70] Stéphane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

[71] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[72] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–4, 2011.

[73] Ashutosh Saxena, Justin Driemeyer, Justin Kearns, and Andrew Y Ng. Robotic grasping of novel objects. In *Proceedings of the 19th International Conference on Neural Information Processing Systems (NIPS)*, pages 1209–1216. MIT Press, 2006.

[74] Ashutosh Saxena, Justin Driemeyer, and Andrew Y. Ng. Robotic grasping of novel objects using vision. *International Journal of Robotics Research*, 27(2):157–173, 2008.

[75] Ashutosh Saxena, Min Sun, and Andrew Y Ng. Make3D: Learning 3D scene structure from a single still image. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 31(5):824–840, 2009.

[76] Ashutosh Saxena, Lawson LS Wong, and Andrew Y Ng. Learning grasp strategies with partial shape information. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, volume 3, pages 1491–1494, 2008.

[77] Luis Sentis, Jaeheung Park, and Oussama Khatib. Compliant control of multicontact and center-of-mass behaviors in humanoid robots. *IEEE Transactions on Robotics*, 26(3):483–501, 2010.

[78] Luis Sentis, Josh Petersen, and Roland Philippsen. Implementation and stability analysis of prioritized whole-body compliant controllers on a wheeled humanoid robot in uneven terrains. *Autonomous Robots*, 35(4):301–319, 2013.

[79] Ruben Smits, H Bruyninckx, and E Aertbeliën. Kdl: Kinematics and dynamics library. Avaliable: `http://www.orocos.org/kdl`, 2011.

[80] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[81] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[82] Kenneth O Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence*

Research (JAIR), 21:63–100, 2004.

[83] Ioan A Sucan and Sachin Chitta. MoveIt! Available: `http://moveit.ros.org`, 2013.

[84] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics and Automation Magazine*, 19(4):72–82, 2012.

[85] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[86] The Human-Centered Robotics Laboratory, Mechanical Engineering Department, The University of Texas at Austin. *Dremer* Robot. `http://sites.utexas.edu/hcrl/`.

[87] Huahua Wang and Arindam Banerjee. Bregman alternating direction method of multipliers. In *Advances in Neural Information Processing Systems*, pages 2816–2824, 2014.

[88] ROS community Willow Garage. ORK: Object Recognition Kitchen. Available: `https://github.com/wg-perception/object_recognition_core`.

[89] Brian G Woolley and Kenneth O Stanley. Evolving a single scalable controller for an octopus arm with a variable number of segments. In *International Conference on Parallel Problem Solving from Nature*, pages 270–279. Springer, 2010.

[90] Keenan A Wyrobek, Eric H Berger, HF Machiel Van der Loos, and J Kenneth Salisbury. Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2165–2170, 2008.

[91] Hao Zhang, Pinxin Long, Dandan Zhou, Zhongfeng Qian, Zheng Wang, Weiwei Wan, Dinesh Manocha, Chonhyon Park, Tommy Hu, Chao Cao, et al. Dorapicker: An autonomous picking system for general objects. *arXiv preprint arXiv:1603.06317*, 2016.

[92] Jianwei Zhang and Bernd Rössler. Self-valuing learning and generalization with application in visually guided grasping of complex objects. *International Journal of Robotics and Autonomous Systems*, 47(2):117–127, 2004.